A Feature Location Approach for Mapping Application Features Extracted from Crowd-based Screencasts to Source Code

Parisa Moslehi
Concordia University
Montreal, Canada
p_mosleh@encs.concordia.ca

Bram Adams
Polytechnique Montreal
Montreal, Canada
bram.adams@polymtl.ca

Juergen Rilling
Concordia University
Montreal, Canada
juergen.rilling@concordia.ca

Abstract

Crowd-based multimedia documents such as screencasts have emerged as a source for documenting requirements, the workflow and implementation issues of open source and agile software projects. For example, users can show and narrate how they manipulate an application's GUI to perform a certain functionality, or a bug reporter could visually explain how to trigger a bug or a security vulnerability. Unfortunately, the streaming nature of programming screencasts and their binary format limit how developers can interact with a screencast's content. In this research, we present an automated approach for mining and linking the multimedia content found in screencasts to their relevant software artifacts and, more specifically, to source code. We apply LDA-based mining approaches that take as input a set of screencast artifacts, such as GUI text and spoken word, to make the screencast content accessible and searchable to users and to link it to their relevant source code artifacts. To evaluate the applicability of our approach, we report on results from case studies that we conducted on existing WordPress and Mozilla Firefox screencasts. We found that our automated approach can significantly speed up the feature location process. For WordPress, we find that our approach using screencast speech and GUI text can successfully link relevant source code files within the top 10 hits of the result set with median Reciprocal Rank (RR) of 50% (rank 2) and 100% (rank 1). In the case of Firefox, our approach can identify relevant source code directories within the top 100 hits using screencast speech and GUI text with the median RR = 20%, meaning that the first true positive is ranked 5 or higher in more than 50% of the cases. Also, source code related to the frontend implementation that handles high-level or GUI-related aspects of an application is located with higher accuracy. We also found that term frequency rebalancing can further improve the linking results when using less noisy scenarios or locating less technical implementation of scenarios. Investigating the results of using original and weighted screencast data sources (speech, GUI, speech and GUI) that can result in having the highest median RR values in both case studies shows that speech data is an important information source that can result in having RR of 100%.

Keywords- *Crowd-based documentation; mining video content; speech analysis; feature location; software traceability; information extraction; software documentation*

1. Introduction

In traditional software development processes, software documentation has played a vital role in capturing information relevant to the various stakeholders and as assessment criteria for the maturity and quality of a software product and its underlying development processes (Moslehi, Adams and Rilling, 2016). However, over the last decade, with the economical (e.g., globalization, open source projects), social (e.g., collaborative and agile work habits), and technological (e.g., Internet) changes in our society, new software processes (e.g., agile and open source) have been introduced to deal with these ongoing changes. What is common to these development processes is that communication and data become more important than formal and complete documentation. As a result of this paradigm shift, projects are now often left with no or very little documentation (Turk, France and Rumpe, 2014), or documentation that is often incomplete or lacking sufficient details (e.g., explanations or examples) to meet the needs of project stakeholders.

Product reviews, how-to videos, tutorials, Q&A sites like Stack Overflow[1] have started to replace many of the more traditional forms of documentation media, requiring open source users and contributors to resort to the Internet for finding relevant documentation and insights to support their current work context. This trend has been further facilitated by the next generation (also referred to as digital natives) of developers entering the workforce, who grew up using the Internet to create and share their own work. This change on how people use and share information on the Internet increased the popularity of crowd-based documents (e.g., Wiki pages, Q&As, mailing lists, tutorial videos) that are created by many and viewed by many (Parnin *et al.*, 2012). However, this ever-increasing volume of crowd-based information and resources has also led to a situation where information resources are now not only fragmented across different online portals and repositories, but also across different types of resources (e.g., textual and multimedia), making it difficult for users to locate and navigate through relevant artifacts.

One type of crowd-based multimedia documents that has gained popularity in recent years, are screen-captured videos (i.e., screencasts). Screencasts deliver their content in the form of audio (narrator), video (images), and textual metadata (e.g., subtitle, title, description). For example, a user might demonstrate how to perform a certain task in an application by clicking on GUI buttons, entering text, etc. according to that user's workflow. Similarly, one could visually demonstrate how a bug or vulnerability could be triggered in an application, or what the new features of an upcoming application release are (Moslehi, Adams and Rilling, 2018). This kind of information contrasts with more traditional software artifacts, where content delivery is based mostly on textual or, at best, static graphical snapshots (instead of a dynamic sequence of snapshots). In fact, the open source community has started to make screencasts an integrated part of their product documentation. For example, WordPress[2] encourages its users and contributors to create new how-to videos[3] that describe certain use-case scenarios or features. Some users even enrich their bug reports through screencasts[4] to exactly demonstrate how to reproduce a bug.

Mining crowd-based multi-media documentation is still an emerging research field (MacLeod, Bergen and Storey, 2017), (MacLeod, Storey and Bergen, 2015). While the MSR community has addressed the problem of mining unstructured repositories such as mailing lists, Q&A sites and Wiki pages by analyzing non-structured free form text, only limited work exists on mining, linking, and analyzing video files. In our previous research (Moslehi, Adams and Rilling, 2018), we presented an approach that leverages high-level information found in both the audio (i.e., speech) and visual content (i.e., image frames) of screencasts to locate application features presented in the screencasts and link them to their corresponding source code implementations. We conducted a case study on 10 WordPress screencasts that showcase how to use a WordPress feature. The results from our case study showed that our approach is capable to locate source code artifacts that are relevant to such screencasts.

In this research paper, we extend our previous work (Moslehi, Adams and Rilling, 2018) with results of a user survey and an additional case study on a larger data set of screencasts for Mozilla Firefox. In addition, while we applied, in our previous work, unigram topic modeling to link screencasts to WordPress source code, we extended these case studies using bigram topic modeling on the same data set to better understand the impact of gram size on the performance of our approach. We also evaluate our linking approach on both WordPress and Firefox' front- and back-end. These additional studies provide us with insights on the need for and performance of our mining approach to establish traceability links between product features described in screencasts and source code artifacts implementing these features. The main **contributions of this paper** are as follows:

- We provide a summary of a survey which we conducted in (Moslehi, Rilling and Adams, 2020) to help support and motivate the research in this paper. We report on some of the relevant survey results that show, screencasts demonstrating application features are actually used by software engineers with different expertise levels and for different purpose.

[1] https://stackoverflow.com/
[2] https://wordpress.com/
[3] https://en.support.wordpress.com/video-tutorials/
[4] https://youtu.be/Am9SNUhSz4w

- We compare the performance of our approach on WordPress using different gram sizes (unigram and bigram).
- We perform a case study on WordPress and Mozilla Firefox to evaluate how our approach is capable to link video content to the relevant source code artifacts.
- We further evaluate and compare the applicability of our approach in linking application features shown in screencasts to their corresponding front- and back-end source code implementation.
- We compare the performance of our approach on Mozilla Firefox with a guided search approach.

The remainder of this paper is structured as follows: In the next section, we describe our motivation. Section 3 gives a brief introduction to crowd-based multimedia documentation and the topic modeling technique that we used in our approach. Section 4 covers related work, where we describe and contrast similar attempts in analyzing and linking crowd-based documentation and text retrieval-based feature location. Section 5 explains our proposed methodology. We then present our evaluation techniques and our experiments and results on the research in section 6 followed by a discussion in section 7. We present the opportunities and threats to validity of our study in section 8 and finally conclude in section 9.

2. Motivation

Research (Wells, Barry and Spence, 2012) has shown that using tutorial videos as a learning aid for traditional lectures can significantly improve students' performance in education. Also, when compared to written text, screencasts can provide a clearer explanation of a subject matter while being accessible through different devices (Mohorovičič, 2012). The latter work also showed that visual learners and students benefit from screencasts by being able to learn at their own pace, whenever and wherever they want (Mohorovičič, 2012).

In software engineering, tutorial videos are used as a medium to share information for different purposes (e.g., learning, technological trends, job training, how-to guides) (Storey *et al.*, 2014). Multimedia documents have started to replace existing structured software documentation and workflows that have been used traditionally to capture and document software projects. Agile development approaches no longer rely only on formal documentation and instead promote more informal types of software documentation.

Our approach is motivated by the fact that, especially for open source projects or agile development processes, existing system documentation is often incomplete, inconsistent or does not provide clear enough examples. Developers and maintainers therefore often resort to the Internet for additional help (Parnin *et al.*, 2012), (MacLeod, Storey and Bergen, 2015). These online resources are created by users who are not directly involved in the development of the actual product, also referred to as "the crowd". Such crowd-based documentation exists in different formats such as multimedia or informal textual documentation, describing a product feature or how to solve a particular problem (Subramanian, Inozemtseva and Holmes, 2014). Furthermore, users may also create a bug report that is enriched by a screencast that clearly showcases the steps of reproducing a bug[5] on the GUI of an application or provide workarounds for an issue[6]. Developers then must locate the source code artifacts that are relevant to the bug to fix the reported issue.

To further investigate the relevance of using tutorial screencasts as an information source in software engineering we conducted a survey in earlier work (Moslehi, Rilling and Adams, 2020). The objectives of this user survey[7] were as follows:

[5] https://www.reddit.com/r/firefox/comments/4fq1g0/firefox_ui_bug/
[6] https://youtu.be/CIvTVvFTWDA
[7] https://mcislab.github.io/publications/2020/emse_parisa/OnTheUseOfMultimediaDocumentationGoogle_Forms.pdf

1) to identify the type of documentation software engineers with different levels of experience prefer to use and in which context (e.g., help with a particular software engineering task, to improve their current skills, or getting familiar with the GUI of a software application);

2) to better understand how frequently they watch how-to tutorial videos and what they consider as key benefits and disadvantages of using tutorial screencasts over written documentation; and

The survey questions comprise multiple choice questions, free-form answers and scenarios that are related to both programming and non-programming purposes (e.g., questions 14, 28, 29, 30, 31 in the survey). In what follows, we summarize some of the results from our complete survey (Moslehi, Rilling and Adams, 2020) to further motivate the work presented in this research.

For the survey, we received 99 responses from participants with different backgrounds in the software engineering domain. There were 61 graduate students and 14 undergraduate students among the participants. Most of the participants stated that they have 2-5 years of professional experience as a software engineer (33.3%) and 19.2% had no prior professional experience as a software engineer (Figure 1). Among the most popular programming languages that we received in the responses are “Java”, “Python”, “JavaScript”, “C++”, “C#”, and “PHP” (Figure 2). Since “JavaScript” and “PHP” are typically used for frontend development, we assume that: a) participants who selected one or both of these languages can be considered as frontend developers, b) participants who selected “Java”, “Python”, “C++”, and “C#” are more likely involved in backend development and, c) participants who chose languages that are in both categories can be considered as both backend and frontend developers. Based on this classification, our further analysis of the received responses showed that 43% of the participants can be considered as only backend developers and 54% are both backend and frontend developers since they use languages that fall into both categories. Another 3% of the participants are working with languages that do not belong to these two categories.

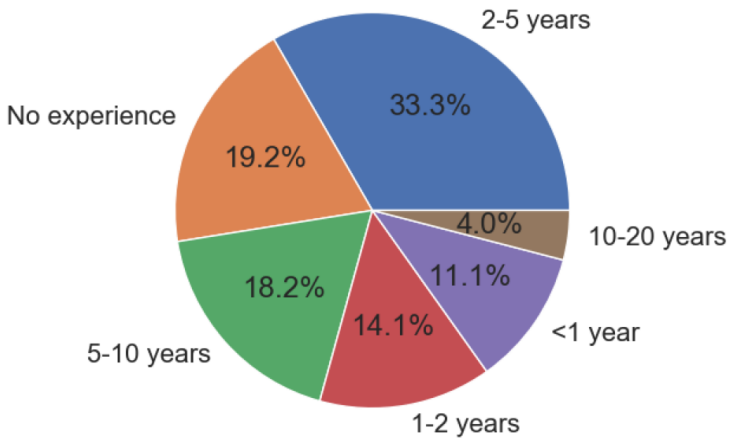


Fig. 1 Professional experience of the survey participants as a software engineer.

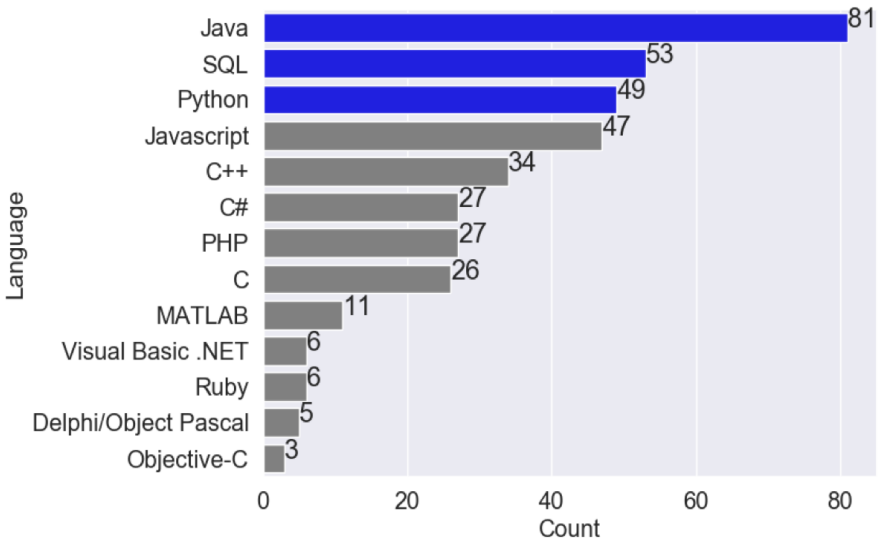


Fig. 2 Programming languages that participants use for development or maintenance tasks. Top 3 languages are colored in "blue".

We summarize the findings from the survey as follows:

The findings show that while tutorial screencasts are considered by most users (i.e., survey participants) to be useful for "comprehending and learning new tasks/concepts", the level of experience plays an important role in choosing a media type or an information source for different purposes/tasks (here we summarize the results for screencasts). Based on the results, "to complete their tasks or learn new skills", survey participants that have less than 2 years of professional experience prefer to use "YouTube" or "Videos" as information sources compared to other social media resources (e.g., GitHub, Stack Overflow, Twitter, etc.) or other media types (e.g., books, blog posts, question and answer sites, etc.). For technical or programming-related purposes (e.g., learning a programming language), survey respondents with less than one year of professional experience mostly watch screencasts. This is while more experienced participants watch videos "to familiarize themselves with an application" or "to learn how to setup an application" which are of more GUI related or non-technical (non-programing) tasks.

Furthermore, survey participants found the ability to trace and link crowd-based screencasts with other software artifacts or documentation to be an important aspect. In response to an open-ended question, survey participants made suggestions about providing traceability to "*keep screencasts up-to-date*" (i.e., version compatibility) or to "*complement screencasts with other documents*". Having traceability between screencasts and other software documentation can help providing more details or documentation that complement the screencast content (e.g., source code), updating screencasts' content (e.g., based on the changes in the code such as adding new features), and locating screencasts that are relevant to the users' information need.

With the assumption of contributing to an open source project in which our survey respondents have no prior experience, 27.3% of the participants indicated that they would often try to locate source code artifacts related to the content they watched in a video (i.e., "**a feature that is being demonstrated in a video**"), with most of this content being GUI or non-technical (non-programming) related video content. Also, 4% declared that they *always* try to do so when they contribute to an open source project. Furthermore, again 27.3% of the participants indicated that they *often* "decide to watch a screencast that showcases application features of a software project whose source code they want to customize" and 5% declared that they *always* try to do so.

These findings show that screencasts are not only a popular source of documentation but also linking them to other documents and application source code is considered by many users to be important for maintaining source code, understanding application features/concepts, and keeping screencasts up to date.

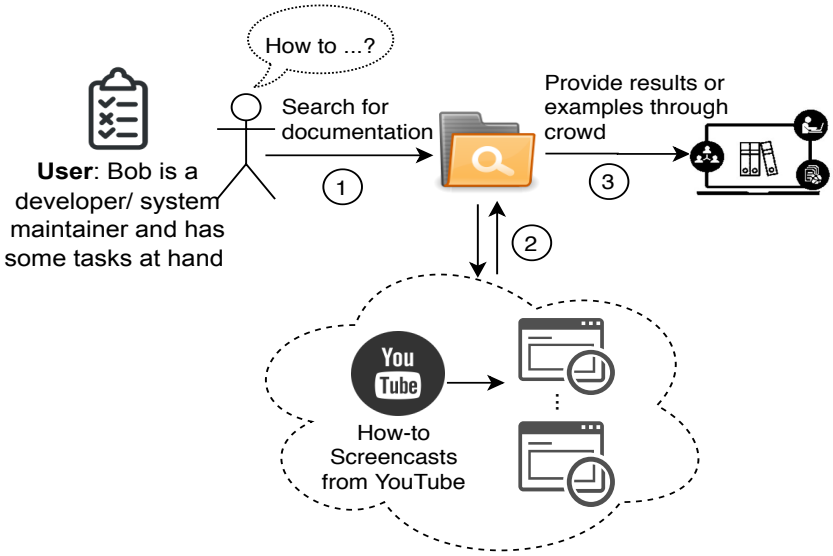


Fig. 3 Motivating example.

Figure 3 shows an example of a fictional web developer named Bob, an open source enthusiast, who is new to WordPress. As part of his contribution, Bob decides to modify a comment moderation feature in the source code of the Akismet WordPress plugin for his personal project. Bob, who is unfamiliar with the workflow and innards of the plugin, first resorts to the available software documentation **to understand the high-level context** and features the plug-in provides to users. However, Bob is unable to find any relevant documentation describing these features and their (sequential) interaction in the existing project documentation. He therefore decides to search the Internet for how-to videos and blogs describing the plugin features. Such documentation, especially how-to screencasts, by many developers are considered to be more intuitive since they visually and dynamically can demonstrate workflows compared to written documentation (MacLeod, Storey and Bergen, 2015).

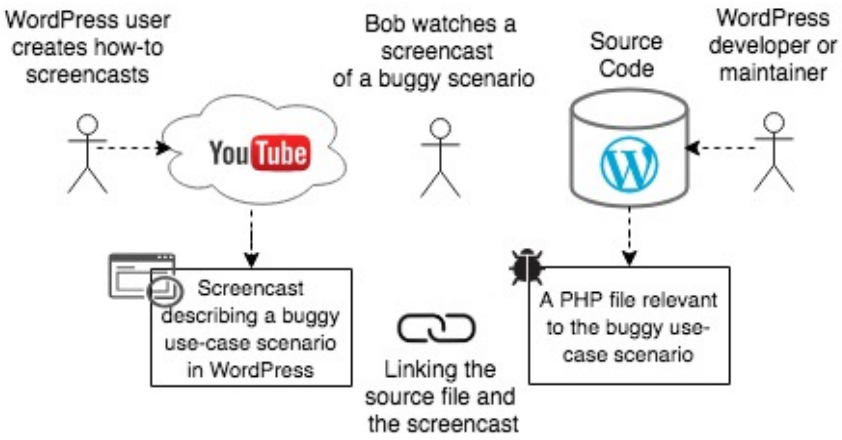


Fig. 4 How linking source code features to screencasts helps WordPress users or developers.

While watching a how-to video, Bob notices that the narrator mentions a bug when setting up the plug-in in WordPress dashboard. Bob decides **to contribute to the WordPress plugin** (Figure 4) by fixing the bug that was shown in the screencast. However, Bob being unfamiliar with the implementation details of the plug-in, finds it challenging to locate the implementation of the buggy feature in the source code.

As the motivating example has illustrated, software developers and maintainers might rely on screencasts that showcase GUI features (not source code), and the workflow used to manipulate those features, for different reasons (e.g., learning how to use application features, how to reproduce a bug). However, users and creators of

screencasts face several challenges when screencasts are being used as a source of software documentation (MacLeod, Storey and Bergen, 2015). We categorized these challenges as follows:

- **Relevance**: How well does a screencast describe the task at hand? Although many screencasts and crowd-based documentation may contain useful information, developers find it still challenging to identify the most relevant video to watch. They have to manually browse through videos to see whether a video contains content that is relevant to their needs or to skip non-relevant parts of videos. While existing search engines provide a ranking of their search results, this ranking typically relies on indexing of user-provided titles and descriptions, or words recovered from automatic speech-to-text but does not consider the actual content of the video.

- **Quantity:** How many videos exist that cover the same topic? Depending on the popularity of the topic, many videos exist that often vary significantly in their length and content.

- **Traceability:** What are the software artifacts that are relevant to the content that is presented in a screencast? While screencasts demonstrate buggy scenarios or workarounds, developers who want to resolve such bugs or integrate the described workarounds into their own application still need to manually trace the screencast content to the application's source code. This process can be time-consuming, especially if a developer is unfamiliar with the project and its implementation.

Considering the increasing popularity of tutorial screencasts, especially among young professionals and students, there is a need to link the content of screencasts (e.g., showcasing how to use certain product features) to different software artifacts to provide support for various programming activities. The main motivation of our research can therefore be described as deriving a methodology for mining the content of screencasts and link this video content to the relevant source code implementing of application features shown in screencasts. More specifically the contributions of our research are:

Derive an approach that takes advantage of content created by the crowd, a resource that has been unavailable in the past and is still underutilized currently by users and organizations, to enrich existing system documentation. More specifically, this includes **extending traditional MSR approaches to mine and analyze video artifacts** (e.g., speech and image frames) related to features of a software product. As part of our research, we introduce a methodology that allows the automated linking of videos to their relevant source code artifacts. Since our goal is not to link source code shown in videos to their source code files, it should be noted that our work differs from that of researchers like Khandwala et al. (Khandwala and Guo, 2018) and Ponzanelli et al. (Ponzanelli *et al.*, 2016, 2019). Indeed, we instead propose an approach that can link content (features) demonstrated in screencasts to the corresponding source code implementation of the demonstrated tool, not the source code shown on screen.

3. Background

Our research is based on concepts and techniques from several sub-domains within computer science, including crowd-based multi-media documentation, topic modeling, traceability and feature location. In what follows, we provide a brief introduction of the underlying concepts and techniques used in this research. If you are already familiar with these concepts, you can safely move on to the next section.

3.1. Crowd-based Multi-Media Documentation

Software users and developers use the Internet to search for informal documentation that can support them during specific tasks. These types of documentation that are created by many and viewed by many are called "crowd-based documents"(Parnin *et al.*, 2012), (Moslehi, Adams and Rilling, 2016). Crowd-based documents

can be categorized as textual (e.g., wikis, emails, Q&As[8]) and multimedia (e.g., images, podcasts, screencasts[9]) documents.

Multimedia documents, and more specifically screencasts, differ from formal and textual documents. Screencasts are "movies of software" that capture the screen as the narrator describes verbally (through speech) and visually (by demonstrating) certain features of a software application or a programming language, by navigating through menu options or the source code respectively (MacLeod, Bergen and Storey, 2017). They are digital documents that are composed of one or multiple media elements (text, image, video, sound, etc.) as a logically coherent unit. The motivation behind creating such documents is for the document creators to gain an online reputation as well as to learn a new subject better and improve their skills by teaching them to others, and to describe features that are difficult to be explained as text only (MacLeod, Storey and Bergen, 2015).

Unfortunately, several issues exist that make it challenging for screencasts to be analyzed and integrated with other types of software artifacts:

- **Abstraction level:** The content representation in a screencast, which is verbally narrated and visually demonstrated, differs significantly from other, more traditional software artifacts. In traditional software artifacts, content is often captured in structured or semi-structured textual formats that simplify the analysis of content across these structured artifacts. Furthermore, even technical screencasts tend to be at a higher level of abstraction, only sparsely providing low-level implementation details of the demonstrated features, which makes it challenging for the developers to locate more low-level software artifacts (e.g., code fragments) that are relevant to the screencast.

- **Dynamic vs. Static:** Screencasts are dynamic by nature, i.e., every couple of image frames the screen changes as the narrator is manipulating for example a GUI widget of the demonstrated software application (Ponzanelli *et al.*, 2016). In contrast, there are other crosscutting software artifacts (e.g., source code, emails, wiki) that are static and relevant to the demonstrated version of the software application. This crosscutting makes it inherently difficult to manually identify and link software artifacts to part of the dynamic content of a screencast that were shown only at some specific point of time during a screencast.

- **Information resources:** Screencasts can contain different information resources (e.g., images, speech, captions, sequence of GUI events, user actions) (MacLeod, Storey and Bergen, 2015), which differ significantly from those found in software artifacts such as source code (e.g., language-specific syntax, comments, function names, identifiers, file names, string literals). It should be noted that the availability and quality of these information resources can differ from screencast to screencast.

3.2. Topic Modeling

Topic models provide statistical information related to sets of words ("topics") that occur together often enough to represent a semantic relation (Blei, 2012). For example, in a newspaper the "sports" topic consists of sports-related words that are semantically related and co-occur across most of the sports-related articles. Topic models can be used in automatic indexing, searching, classifying, and structuring a large corpus of text (Chen, Thomas and Hassan, 2016).

Generally, topic modeling has been used in different software engineering tasks such as document clustering (Kuhn, Loretan and Nierstrasz, 2012), (Kuhn, Ducasse and Gîrba, 2007), feature location (Baldi *et al.*, 2008; Bassett and Kraft, 2013), bug prediction (Nguyen *et al.*, 2012), source code evolution (Thomas *et al.*, 2010), traceability (Asuncion, Asuncion and Taylor, 2010), (Ali *et al.*, 2012), and search (Grechanik *et al.*, 2010),

[8] Portals such as https://www.wikipedia.org/ and https://stackoverflow.com/ contain crowd-based textual documentation.
[9] Portals such as https://commons.wikimedia.org/wiki/Main_Page and https://www.youtube.com/ contain crowd-based multimedia documents.

(Bajracharya and Lopes, 2012). The repositories whose data has been used as input for topic modeling include source code, email, requirements and design documents, logs, and bug reports (Chen, Thomas and Hassan, 2016).

Latent Dirichlet Allocation (LDA) is a probabilistic topic model proposed by Blei et al. (Blei, Ng and Jordan, 2003). LDA is a fully generative model that works in a way that assumes the corpus contains a set of *K* topics. The topics are corpus-wide, which means that each document of the corpus contains one or more topics that describe the entire corpus. Also, each term in the entire corpus vocabulary can be contained in more than one topic and each term in a document originates from one single topic. The most important benefit of LDA is that, since topics are corpus-wide and generated based on all available documents in the corpus, topics for newly added documents can easily be inferred. Also, LDA overcomes the statistical shortcomings of LSI, such as the assumption that the term counts in the corpus follow a Gaussian distribution (Hofmann and Thomas, 2001). In addition, LDA generates human-readable topics that can be used and interpreted easily. As a result, LDA has become a popular topic modeling approach. In Chen et al. (Chen, Thomas and Hassan, 2016), the authors provide an overview of other variants of LDA (e.g., Hierarchical Topic Models (HLDA) (Blei *et al.*, 2003), Supervised Topic Models (sLDA) (Mcauliffe and Blei, 2008), Labeled LDA (LLDA) (Ramage *et al.*, 2009), etc.).

3.3. Software Traceability

Software traceability has been widely recognized as an important quality factor of well-engineered software systems (Cleland-Huang *et al.*, 2014). Software and system traceability can be defined as the ability to establish links between related software artifacts to maintain safe and correct operation of critical software systems (Cleland-Huang *et al.*, 2012). Feature location is a form of traceability that automatically discovers the source code artifacts that implement a certain feature of a software application (Chen, Thomas and Hassan, 2016). It aims to find the starting point (seed) in the source code that corresponds to a system functionality to guide manual exploration (Dit *et al.*, 2013). Using such a seed will reduce the effort required by a software maintainer to locate the parts of the code relevant to a particular feature.

Software traceability can help stakeholders to discover discrepancies and inconsistencies between requirements and their implementation, assessing if software requirements are completely covered in the implementation, and is typically required to receive certificates of assurance (Gotel *et al.*, 2012), which are certificates that ensure the software conforms to user requirements or other policies (Gaffney Jr., 1981).

For open source projects, it has been shown that the requirements analysis and requirements traceability of a project is very different from that of traditional approaches in software engineering (Kagdi and Maletic, 2007). There are different types of informal resources that capture requirements and documentation of a project, such as issue trackers, emails, source control repositories, and screen-captured videos. To reduce maintenance effort, it is essential to keep such project documents consistent with the current state of the source code by being able to uncover traceability links between these artifacts (Kagdi, Maletic and Sharif, 2007).

4. Related Work

In what follows, we present an overview of closely related work, as well as a discussion on how our methodology differs from this existing research. We first describe research related to linking and analyzing crowd-based documents followed by a review of text retrieval-based feature location approaches. We then present current work on analyzing software engineering screencasts.

4.1. Linking and Analyzing Crowd-based Documents

A significant body of work exists that attempts to establish traceability links between crowd-based documents such as Q&A sites (e.g., (Jiau and Yang, 2012; Barzilay, Treude and Zagalsky, 2013; Subramanian, Inozemtseva and Holmes, 2014)) and source code artifacts. For example, Jiau and Yang (Jiau and Yang, 2012) measured the

inequality of crowdsourced API documents in StackOverflow and found that a larger proportion of existing documents addresses a smaller portion of topics and that the documents obeyed the Pareto principle or 80:20 rule. They leveraged this inequality and proposed a method that projects a crowdsourced documentation into a concept domain and recovers traceability links based on reusability of the concept domain. They claim that their proposed method improves documentation coverage by 400%, when they reuse existing documentations that are related to popular API concepts for unpopular API concepts. Barzilay et al. in (Barzilay, Treude and Zagalsky, 2013), also explored the design and characteristics of StackOverflow and developed a code search tool, *Example Overflow*, on top of StackOverflow to extract high quality code examples. As part of their empirical analysis, they studied the type of questions posted on StackOverflow and to what extent these questions could be answered by their approach. Subramanian et al. (Subramanian, Inozemtseva and Holmes, 2014) proposed a method for linking code examples posted on StackOverflow to API documentation. Based on the proposed method, they implemented a tool, *Baker*, that links code snippets to Java classes and methods or JavaScript functions, with an observed precision of 97%.

Many exploratory studies have been conducted to analyze the characteristics of textual crowd-based documents (e.g., (Parnin *et al.*, 2012), (Campbell *et al.*, 2013), (Pham *et al.*, 2013)). Parnin and Treude (Parnin *et al.*, 2012) investigate in their work the use of crowd-based documents to replace traditional software documentation. They measured in their work the extent to which blog posts cover methods of a particular API. They observed that social media posts can cover 87.9% of the API methods and therefore could potentially replace traditional documentation. Nasehi et al. (Nasehi *et al.*, 2012) analyzed code examples of Stack Overflow that are voted by users as being good code examples to identify the characteristics that, if applied, can improve the development and evolution of API documentation. Campbell et al. (Campbell *et al.*, 2013) combined Stack Overflow questions and PHP and Python projects' documentation and used LDA and found topics in Stack Overflow that did not overlap the topics on projects' documentation. Their results also show that many topics that are covered on Stack Overflow but not by traditional project documentation, are related to external project documentation or tutorials. Pham et al. (Pham *et al.*, 2013) investigate the possibility of extracting a common testing culture to tackle challenges that social coding sites introduce to testing behavior. They conducted a survey among GitHub users to identify the challenges, the impact of these challenges on testing practices and based on the survey results they suggest strategies that, if applied by software developers and managers, can positively affect the testing behavior in their projects.

While this existing research analyses how developers can use textual content extracted from crowd-based documents as a source of documentation, our work focuses on the extraction and integration of crowd-based *multimedia content* with source code.

4.2. Text Retrieval-based Feature Location

Several feature location techniques have been proposed in the literature, with a detailed overview of these techniques being presented in (Dit *et al.*, 2013). In what follows, we discuss the work that is most closely related to our approach.

Marcus et al. (Marcus *et al.*, 2004) leveraged Latent Semantic Indexing (LSI) (Deerwester *et al.*, 1990) to find semantic similarities between a query that is either automatically generated or provided by the user to locate source code features. Van der Spek et al. (van der Spek, Klusener and van de Laar, 2008) also used LSI to locate features in source code. They applied different preprocessing and text normalization methods, such as stemming, stop words removal, common terms weighting to their data sets and then evaluated effect of each processing step on their feature location approach.

To improve feature location results, Bassett et al. (Bassett and Kraft, 2013) proposed a new term weighting technique that uses the structural information in the source code (i.e., function names and method calls extracted from call graphs). Since tf-idf term weighting is designed for unstructured documents written in natural language, they overcome this limitation by applying tf-idf in source code-related text using an LDA-based approach. Their

evaluation shows that the LDA-based approach delivers more accurate results. Eddy et al. (Eddy, Brian P. and Kraft, Nicholas A. and Gray, 2018) expanded the work by Bassett et al. (Bassett and Kraft, 2013) by exploring a broader range of criteria in the source code (e.g., leading comments, method names, parameters, body comments, and local variables) and weighted them according to their position in a method. Lukins et al. (Lukins, Kraft and Etzkorn, 2010) proposed an LDA-based bug localisation technique using queries that are extracted and formulated out of bug reports. They evaluated their approach by calculating the percentage of bug queries whose first relevant result is in the top 10 or top 1,000 results.

Common to these existing approaches is that they use LDA or LSI to locate software artifacts through queries, which are either provided by a human or automatically generated from information found in bug reports or source code data sets. In contrast, our approach applies LDA to extract high-level information about features extracted from screencasts and then links this information to the implementation of these features. Furthermore, to improve our linking results, we apply term weighting to modify the term frequencies in screencasts using corpus-wide term frequencies.

4.3. Linking and Analyzing Software Engineering Screencasts

The first study on using crowd-based screencasts to share and document developer knowledge was conducted by MacLeod et al. (MacLeod, Storey and Bergen, 2015), who investigated why developers create screencasts. They also discuss the benefits and challenges of this type of knowledge sharing by analyzing 20 tutorial screencasts and interviewing 10 developers/YouTubers. They found that by using screencasts developers demonstrate and share information related to how to customize a program, the challenges they encountered and their development experiences, solutions to problems, how to apply design patterns, and their programming language knowledge. They also observed that developers are creating these screencasts to promote themselves and gain reputation by helping others. MacLeod et al. (MacLeod, Bergen and Storey, 2017) performed a study of Ruby on Rails screencasts comparing screencasts published through free platforms (i.e., YouTube) with screencasts specifically designed for a specialized paid platform (i.e., RailsCasts). As part of this study, the authors extract guidelines for screencast creators on how to produce clear and understandable videos.

Other research studied the usage of crowd-based tutorial screencasts in the software engineering domain. Amongst the earliest work in this area is that of Bao et al. (Bao *et al.*, 2015, 2017), who developed a video scraping tool, *scvRipper*, to automatically extract developers' behavior from screencasts. They extract actions of a developer by employing key point-based template matching based on visual cues in an image (e.g., icons that appear in a window). Although *scvRipper* is not sensitive to screen resolutions and window color schema, the requirement for applications in screencasts to use the same layout and window structures reduces the generalizability of the approach. Bao et al. [45] proposed a method to track user activities and developed a tool called *ActivitySpace* to support inter-application information needs of software developers. The tool reduces the effort required by developers for locating documents and recalling their history activities in daily work.

In (Bao *et al.*, 2019), Bao et al. also introduced a tool, *VT-Revolution*, that captures the workflows in programming tutorials and enables timeline-based browsing of tutorials as well as accessing the API documentation of a selected code element. In their approach, they record the low-level Human-Computer Interaction (HCI) data used by tutorial creators while working with screencast authoring tools. Khandwala et al.'s *Codemotion* (Khandwala and Guo, 2018) used a computer vision algorithm to extract source code and dynamic code edit intervals from tutorial screencasts. The tool uses a video player UI to enable code search and navigation. In our previous work (Moslehi, Adams and Rilling, 2016), we extract documentation from the speech component of tutorial screencasts that describe how to use the features of a GUI application. A case study evaluating how this approach can be used to extract usage scenarios from a total of 25 WordPress screencasts of 5 scenarios showed that the approach extracts usage scenarios from screencasts with the median precision and recall of 83.33% and 100%, respectively.

Other work (MacLeod, Storey and Bergen, 2015; MacLeod, Bergen and Storey, 2017), (Ponzanelli *et al.*, 2016, 2019) addresses the need for designing concise video tutorials that have less noise and precisely describe what

viewers need (Ponzanelli *et al.*, 2019). Ponzanelli et al. (Ponzanelli *et al.*, 2016, 2019) developed an approach to extract relevant fragments of software development tutorial videos and link them to relevant Stack Overflow discussions by mining the (captioned) speech and GUI content of the video tutorials. Yadid et al. (Yadid and Yahav, 2016) developed an approach to extract code from programming video tutorials to enable deep indexing of these videos. Their approach consolidates code across multiple image frames of the videos and uses statistical language models to make corrections on the extracted code. Another work by Ott et al. (Ott *et al.*, 2018) presented an approach that uses deep learning, and more specifically convolutional neural networks and autoencoders, to identify source code examples in image frames of a large data set of videos.

Escobar-Avila et al. (Escobar-Avila, Parra and Haiduc, 2017) and Parra et al. (Parra, Escobar-Avila and Haiduc, 2018) presented text retrieval-based tagging approaches to help users to identify whether or not the content of a video tutorial might be relevant to the needs of a user. Poche et al. (Poche *et al.*, 2017) classify tutorial video comments using Support Vector Machines (SVM), to summarize the comments for content creators. Ellmann et al. (Ellmann *et al.*, 2017) used a frame similarity approach (i.e., cosine similarity and LSI) to identify and distinguish development screencasts from other types of videos on YouTube and link them to their relevant API documentation using only the audio transcript of the videos.

Common to this related research is that they use programming tutorial videos as input and try to extract source code or programming documentation from them and link these code artifacts to corresponding source code artifacts. The approaches often rely on a combination of natural language processing (NLP), image processing, and machine/deep learning techniques. In contrast to this existing work, our goal is to analyze and link screencasts that do not show source code and instead demonstrate or provide walkthroughs, using features/options which are typically part of the GUI of the actual application. In our approach we link this high-level GUI information of an application that is shown in a screencast to the corresponding source code artifacts in the software project using LDA.

5. Methodology for establishing traceability links between screencasts and source code

Given a set of screencasts demonstrating a given application feature, our objective is to locate the source code artifacts related to the GUI interactions (scenarios) shown in screencasts. It should be noted that our 4-step methodology (Figure 5) is independent of the programming language used by the application. Before outlining each of these steps in more detail, we first describe the types of data sources used by our methodology.

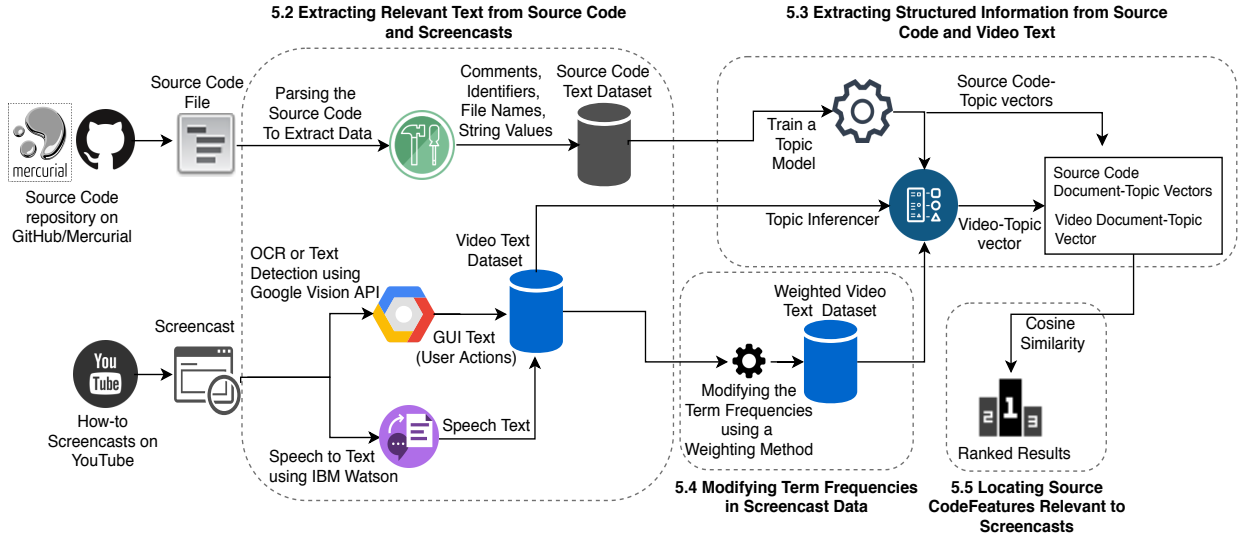


Fig. 5 Methodology for feature location from screencast videos describing an application feature to the corresponding source code implementing this feature.

5.1. Data Sources

We use two data sources for our approach: data extracted from screencasts and source code. As explained by MacLeod et al. (MacLeod, Storey and Bergen, 2015), there are various important elements of information inside screencasts. In what follows, we discuss their potential use for feature location:

Speech: Tutorial screencasts often have a narrator who explains each step of a feature (use-case scenario), while demonstrating the feature on-screen. Parts of the speech not only will match labels or content in the image frames but likely also keywords found in the source code. Figure 6 shows an example of such a keyword matching between screencast content (top) and source code (bottom).

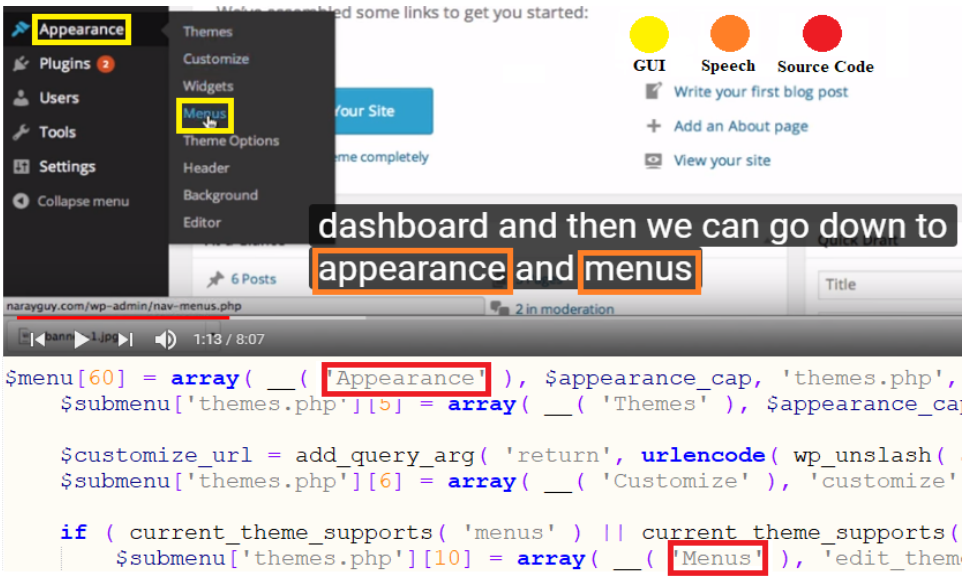


Fig. 6 Matching keywords between speech, GUI and source code.

- *GUI frames*: Screencasts capture GUI interactions within a software application, with each screencast containing a long sequence of images (frames) played at a constant frame rate. Since a typical GUI contains both textual and graphical information, the GUI text on a screencast frame could be matched to corresponding string literals in the source code, unless the content is generated dynamically at run-time (e.g., user input). Furthermore, icons and graphical information such as edges, layout of the visual content and color changes that occur in an image frame could be used as visual information clues to identify which feature aspect the narrator currently is focusing on.

- *User actions:* A tutorial screencast includes textual, graphical and speech information that dynamically changes as a result of a narrator's actions during the screencast (e.g., pressing a button, clicking on a menu item or opening another window). These actions are often closely related to corresponding source code artifacts, such as GUI widget labels or event handlers.

- *Sequence of events:* The order in which user actions occur during a screencast form a sequence of events that could correspond to a chain of event handler invocations or a method call graph.

- *Metadata:* When screencasts are uploaded to video portals like YouTube, they are typically annotated with some form of meta- and viewer related data (e.g., title, description, upload date, comments, number of likes and dislikes, and other information related to the screencast). Such metadata provides additional information that can be used during the linking of screencasts and source code content.

Our second data source contains the source code of the software release exercised in the screencasts. In contrast to the screencast data set, the textual code information is easy to extract from a software project's version

control system and contains mostly low-level information such as source code text and comments. A main challenge when dealing with such low-level information is to extract and abstract meaningful semantic information, to reduce the semantic gap between the vocabulary used by screencasts and by the source code. To close this semantic gap, different elements of source code information should be used to recover the semantics of the developers' objectives (Adrian *et al.*, no date; Kagdi and Maletic, 2007; Subramanian, Inozemtseva and Holmes, 2014; Bao *et al.*, 2015), such as:

- ***Source code comments:*** Comments explain or annotate parts of the source code, usually in higher-level terms than the code itself.
- ***Variables and identifiers:*** Variable names usually relate to the scenario-related information stored within them.
- ***String literals:*** Static string literals often appear on GUI widgets.
- ***Method and class names:*** Developers choose class names and method names closer to the software's domain. These names may also map to the features and their relevant text that appear on the GUI of the application.
- ***File names:*** File names are another useful source of information for feature location, since they typically describe the objective or usage of source code.

5.2. Extracting Relevant Text from Source Code and Screencasts

Source Code Preprocessing: For every source code file, we extract source code elements using Exuberant Ctags[10]. From the full path to a source code file, we extract only the file name (e.g., "browser-customization" from "/path/to/browser-customization.js"). These source code elements are further processed by removing noise in the data, such as special characters, numbers, punctuations and stop words. For the remaining tokens, we perform identifier splitting using '_', camel case and word splitting (e.g., words that contain special characters, such as '-' in 'menu-item' are replaced by a space), as well as word stemming to further normalize and improve the later linking to relevant code elements.

Screencast Preprocessing: Screencasts are composed of an audio component and image frames. In what follows, we describe how we extract the speech and GUI data.

Speech: Tutorial screencasts usually have a narrator who explains the steps involved in performing a certain feature (scenario). For screencasts with closed captioning available, tools (e.g., youtube-dl[11]) can be used to extract a screencast's subtitles. Since not all screencasts have closed captioning available, for all screencasts (published with or without closed captioning), we used automatic speech recognition tools to transcribe the speech information (Moslehi, Adams and Rilling, 2016). Once the transcribed text is available, the same preprocessing steps as for source code are performed.

Image processing and user actions: To extract text from video frames, one needs to first extract image frames of the screencasts, then extract text fragments from them that are relevant to the use-case scenario. Such relevant text fragments typically are either found in the image frames or in text/labels associated with them. Given the many frames within a typical video, only those in which a major event (e.g., mouse click or pressing a button) happens, i.e., key image frames, should be targeted.

Unfortunately, identifying these text fragments and frames is quite challenging. We experimented with three different image processing approaches, before settling with a simpler, textual approach. First, we used Template Matching (Brunelli and Poggio, 1993), where one should provide a template image of a mouse pointer (both the clicking and idle version) to be able to automatically locate the mouse pointer in video frames and any mouse

[10] http://ctags.sourceforge.net/
[11] https://youtube-dl.org/

actions being performed. We also experimented with image pixel subtraction (Nixon and Aguado, 2012a), which subtracts the pixels of neighboring frames to detect changes occurring between image frames, typically caused by mouse movements. Finally, we also used connected components detection (Nixon and Aguado, 2012b), which exploits the fact that neighboring pixels typically have similar pixel values to locate different logical areas in the image frames (e.g., a "text field", or "button").

However, these image-processing approaches have several drawbacks, namely processing overhead, strong dependence on image quality or image resolution, and possible information loss due to image binarization or transformations. We also experimented with both, the transformation of images to gray scale and their binarization to: 1) reduce the dependency of our approach on background color, 2) to improve the recognition of text and icons, and to 3) reduce the computational resource overhead by reducing the image dimensions. However, these transformations caused areas with light background and white text to vanish. In addition, our template matching technique did not perform well due to users customizing their mouse pointers and mouse pointers often being hidden by labels during the mouse click.

Instead, we opted for a much simpler, pure textual approach to detect features during a screencast. For the feature detection we use Optical Character Recognition (OCR) (Cheriet *et al.*, 2007), which we applied on image frames to recognize text shown in the frame. We then subtract the text (instead of the pixels) of every 2 subsequent image frames after extracting image frames with a specified frame rate:

$$\begin{aligned} &Diff(i) = Text(img_i) - Text(img_{i-1}) \\ &Diff = [Diff(2), Diff(3), \dots, Diff(k)] \\ &Diff_{final} = [d_i |\, d_i \in Diff \ \wedge\ |d_i| > 0] \end{aligned} \qquad (1)$$

where k is the number of the image frames of a screencast. OCR returns all text enclosed within neighboring pixels on the images along with its coordinates in the images. An advantage of this approach over image processing approaches is that during the analysis only a bag of words is used and therefore significantly reduces the processing overhead. For example, if a user clicks on a button, the GUI will change to open a dialog menu or new text field, which will be reflected as a change in the text recovered through OCR. $Diff(i)$ contains the bag of words (Figure 7) that are added/modified in each subsequent image frame (i.e., $Text(img_i)$) and are considered to be relevant words to the scenario. Our screencast data sets, in which each screencast becomes a text document, contains only the $Diff_{final}$, which corresponds to the text extracted based on the GUI text difference between successive image frames (formula 1).

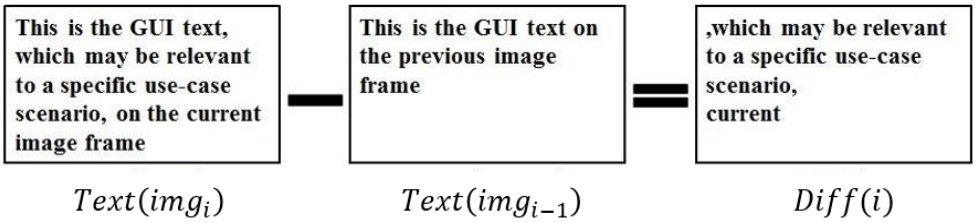


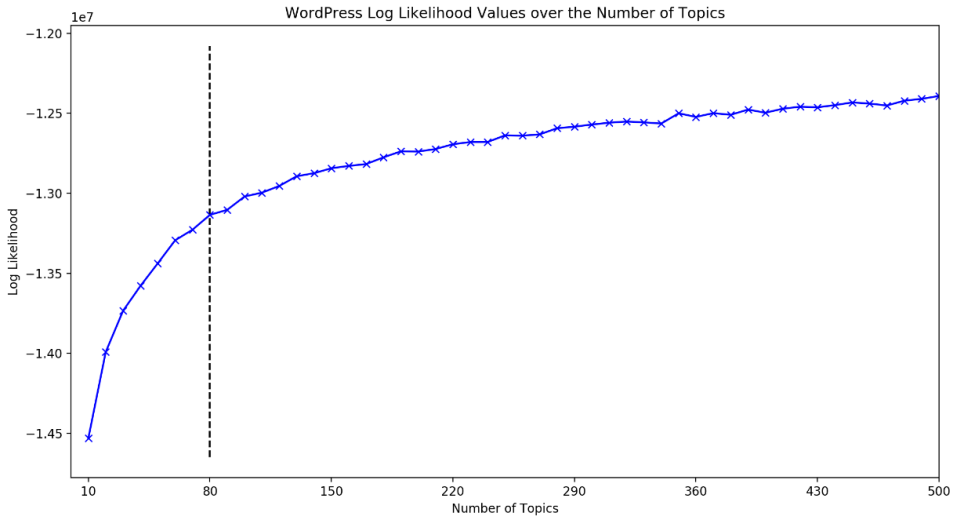
Fig. 7 GUI text difference after a user action happens.

5.3. Extracting Structured Information from Source Code and Screencast Text

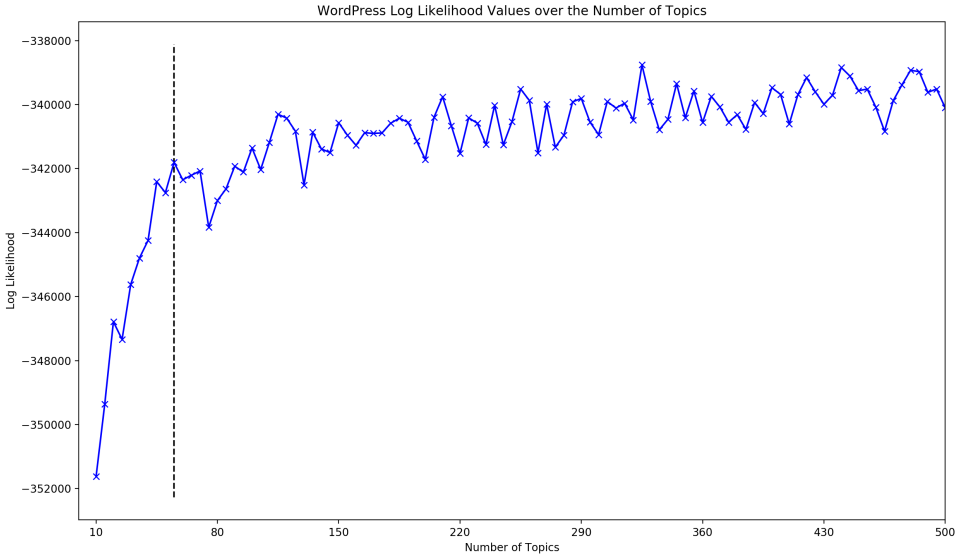
To extract additional semantic structures from the textual representation of the source code and screencast data, we use LDA (Blei, Ng and Jordan, 2003) as a topic modeling approach. As described in section 3.2, in LDA each latent topic is characterized by its statistical distribution over a bag of words, and documents are represented as random mixtures over these topics. LDA transforms each document into vectors of topic probabilities, which are then compared with each other. In our case, documents are either source code files or text files containing information extracted from a screencast (GUI, speech or both), with all extracted text being stored in the same document.

To use LDA, one has first to determine the number of topics that should be used for the topic modeling process. The smaller the corpus, the fewer topics should be generated, while for a larger corpus more topics can be generated (Lukins, Kraft and Etzkorn, 2010). To find the optimal number of topics, we applied the approach used by Thomas et al. (Thomas, 2012), who suggest using different numbers of topics and evaluating the resulting models based on their topics' log likelihood values. The optimal number of topics can be determined by the point where the log likelihood values start to get diminishing returns (i.e., a "knee" in the corresponding plot), which represents a good balance between topic richness and overspecialized topics and avoids having too few or too many topics. As an example, Figure 8 shows the corresponding plots and knees for the two systems (i.e., WordPress and Firefox) used in our case studies.

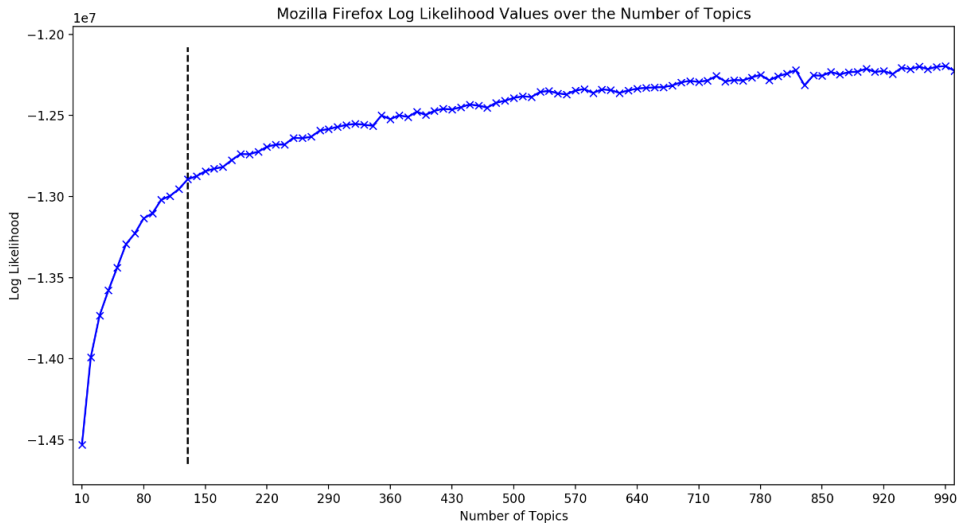
We also can specify the gram size for topic modelling. For instance, bigram topic models split the text into word groups of length 2 (e.g., 'add_new', 'blog_post', 'search_engine', 'remove_password'). Using bigram topic modeling, the model considers the sequential order of words (after the removal of stop words) by exploiting that word *A* was immediately succeeded by word *B* (Wallach, 2006). As a result, words that rarely occur in a corpus (e.g., due to errors and noise from STT and OCR tools) will have very low co-occurrence in bigrams.



a. WordPress – 80 topics using unigram topic modeling



b. WordPress – 55 topics using bigram topic modeling



c. Firefox – 130 topics using bigram topic modeling

Fig. 8 Log likelihood values vs. Number of Topics (K) using different gram size.

ln Jurafsky et al. (Jurafsky and Martin, 2009), the authors performed a comparison of statistical language models using different n-gram models. Their study shows that bigram models result in lower perplexity (higher log-likelihood values) and therefore better models compared to unigrams. As Figures 8a and 8b show, when we compared the log-likelihood values between n-gram models for our WordPress data set, our analysis also showed that bigram topic modeling (with vocabulary size of 8,014 words before applying text preprocessing) outperforms unigram topic modeling. Bigram topic models are considered to be more useful for understanding the semantics of a text (Wang, McCallum and Wei, 2007), since the meaning of some words may be affected by their precedent or subsequent word in a sentence (e.g., 'search' and 'engine' vs. 'search_engine'). Specifying the gram size can be used to adjust the granularity level of the topics as well as reducing the effect of noise in the text and errors in both OCR and STT outputs. As Figure 8a and 8b show, using bigram topic modeling the number of topics will be reduced from 80 to 55 topics for WordPress source code dataset. For projects, such as Mozilla Firefox, where one has to deal with a large vocabulary size (51,294 words before preprocessing the text), multiple programming languages, and different abstraction layers in the implementation, using bigrams can further improve the interpretability of the documents that contain the same terms in different contexts or scenarios.

After determining the number of topics using the source code data set, we then train a topic inferencer using our source code corpus. The topic model creates document-topic vectors from the source code documents. Each element of these vectors corresponds to the probability that a given topic occurs in a document. Next, to be able to use screencast documents to query our source code data set, we use the trained topic inferencer to infer the topics of an input screencast document. We infer screencast documents topics using our source code data set's topic inferencer to convert the screencast documents to document-topic vectors that contain the same topics as our source code document-topic vectors. In the next section, we discuss another preprocessing step which we apply to our screencast documents to modify their term frequencies and see how this can improve our results.

5.4. Modifying Term Frequencies in Screencast Data

The quality of a topic model also depends on the data being used as input. During our initial evaluation of the topic modeling approach, we observed that even after preprocessing, the screencast data contained a significant amount of noise. For a single screencast document containing GUI, speech or both types of data, certain words might be less important to the use-case scenario under study yet occur more frequently than words that are essential to this scenario.

To address this imbalance, we rebalanced the occurrences of terms within documents based on their importance, using a weighting algorithm to modify the term frequencies in a screencast document. The commonly used tf-idf (term frequency-inverse document frequency) (Manning, Raghavan and Schütze, 2008) approach does not apply in our case. In a preliminary study using tf-idf term weighting on our screencast data, we observed the tf-idf did not improve the performance of our approach. This is because tf-idf reduces the weight of frequent terms in favor of words that are rare across the whole corpus. However, in our case, since we use LDA, which is based on a corpus-wide co-occurrence frequency of the terms, these corpus-wide frequent terms are therefore relevant for locating features.

We instead applied a rebalancing approach that favors screencast documents that contain terms that are more likely to be relevant to source code documents. This rebalancing approach allows us to significantly reduce the effect of noise (i.e., non-relevant terms for a given scenario, OCR and STT errors) in each document, when combined with bigram topic models. Our rebalancing approach calculates the term-frequency ranks in a single document based on the corresponding term-frequency ranks in the whole corpus across all available documents of the same use-case scenario. For the rebalancing, we first merge all screencast text documents, each of which containing the $Diff_{final}$ value (formula 1) of a screencast related to the same use-case scenario, into one single document, then calculate term frequencies for this single document (containing all screencasts related to the same scenario). Using these term frequencies, we can now determine the relative importance of each term and rank these terms in each single document based on their assigned corpus-wide term frequency. During the last step of our rebalancing, we modify the term frequency of each term j that appears in each screencast document using the following formula:

$$New_{TF_j} = round\left(TF_j \times \frac{1}{New_{r_j}}\right) \qquad (2)$$

Using the $round()$ function, we round off the New_{TF_j} to its nearest integer, with New_{r_j} being the new local rank of the term j in a single document based on its corpus-wide rank. Using this approach, the term frequencies in each document will be modified to adjust their weight based on the importance of terms across all screencasts for a given scenario. At the same time, lesser or not important terms will have lower term frequencies or are removed completely from the document (i.e., $New_{TF_j} = 0$). For example, given a non-relevant term that is ranked #10 for the whole corpus, and is the most frequent term in a given document (e.g., $TF_j = 35$ and $r_j = 1$). After applying our term weighting approach, its new local rank (New_{r_j}) in the document will be reduced for the document and accordingly its New_{TF_j} will be modified proportional to its importance or new local rank (e.g., $New_{r_j} = 5$ and $New_{TF_j} = 7$). The value of New_{r_j} or the new local rank of a term, shows how many of the other terms that are ranked above the current term (i.e., their corpus-wide rank is higher) exist in the screencast document. In the above example, the corpus-wide rank of the term was #10 and its new local rank is #5, this means that, 5 out of 10 terms did not exist in the screencast document, while they appear in other screencasts.

The rebalanced screencast documents are also used as input to our topic inferencer, which was created using our source code data set (see Section 5.3), to be converted to screencast document-topic vectors and used as our query vectors.

5.5. Locating Source Code Features Relevant to Screencasts

For the last step of our methodology (Figure 5), we use the screencast document-topic vectors as queries and source code document-topic vectors as corpus, to identify source code file(s) that are most relevant to a given screencast. We use the cosine similarity measure (Manning, Raghavan and Schütze, 2008) to compare the screencast vector with each source code vector. The highest ranked vector corresponds to a file with the highest similarity score, which is therefore the most relevant for a given screencast. Eventually, a list of source code files is obtained ranked from most to least relevant for a given screencast. These ranked files can now be used as a starting point (seed) for further source code navigation during the feature location process.

6. Case Studies

In this section, we evaluate our feature location methodology using two distinct data sets to address the following research questions:

- **RQ0.** What is the performance of the approach using unigram topic modeling vs bigram topic modeling?
- **RQ1.** How accurately can source code files be located from screencasts?
- **RQ2.** How does applying our term weighting approach on GUI and/or speech data affect the feature location performance?
- **RQ3.** Are both speech and GUI text data required for feature location?
- **RQ4.** What is the performance of the approach in locating source code related to frontend and backend implementation of a project?
- **RQ5.** How accurately can our approach locate source code directories compared to a guided search approach?

6.1. Case Study Setup

The goal of our case study is to gain new initial insights on how the linking of GUI features/elements shown in screencasts to the source code artifacts implementing these features might be affected when these source code artifacts either belong to the backend or frontend of an application. Additionally, we study how the project size and architecture (with WordPress being a medium-sized and Mozilla Firefox a large project) may affect our linking results. We therefore measure the accuracy of our feature location approach using both:

1) a project (WordPress) whose backend and frontend components[12] closely interact with each other and are also mostly implemented using a single programming language
2) a project (Mozilla Firefox) that follows a stricter multilayered architecture, with frontend and backend not only being decoupled from each other but also being implemented using different programming languages[13].

Data Preparation: For our case studies, our target tutorial screencasts are videos in which a narrator explains the use of an application to perform a specific task. For our study, we considered screencasts for two applications, i.e., WordPress and Mozilla Firefox. Both projects differ significantly in their application domain, their size and architectural design (e.g., their front- and backend implementations). Given the popularity of these projects, for each of them many tutorial screencasts have been created and made available on YouTube that explain how to use the features of these projects. Furthermore, at the time of conducting this research, WordPress is ranked as the best CMS tool[14] in terms of providing tools, variety of themes, and affordable cost. It is also known as the most popular CMS tool used by more than 60 million websites[15]. Mozilla Firefox is known to be one of the top browsers for its speed and security[16] and is the second most popular browser after Google Chrome[17].

A common challenge when analyzing screencasts is that video quality (resolution) differs among screencasts, which will affect the OCR processing. Since our objective is not to evaluate the quality of the OCR, but rather the quality of our linking approach, we consider only High Definition (HD) videos for our study. If a screencast has a narrator, describing the features being displayed during the screencast, only English narrations are extracted and used. In addition, we only considered screencasts for popular features (or scenarios) that are available in the same WordPress or Firefox releases.

[12] https://codex.wordpress.org/images/2/20/WP_27_modules.JPG
[13] http://tiberius.byethost13.com/pcw_lab/lab1/assign1.pdf?i=1
[14] https://www.techradar.com/news/best-cms-of-2018
[15] https://en.wikipedia.org/wiki/WordPress
[16] https://www.toptenreviews.com/best-internet-browser-software
[17] https://en.wikipedia.org/wiki/Firefox

In our approach, as explained in Section 5.3, we train a topic inferencer using the source code then each single video of the same scenario is used to infer its topics (screencast document-topic vectors). The resulted screencast document-topic vectors are used as queries and their similarity to source code document-topic vectors are calculated to retrieve and rank the most similar source code documents. As a result, for the purpose of evaluating the results against different baselines (e.g., performance of the approach on frontend vs. backend in Firefox or *All* vs. *Unique* in WordPress) we use multiple videos since the number of data points in our evaluations (see boxplots in Section 6) directly depends on the number of available screencasts. In addition, the number of available screencasts (and therefore image frames) for each scenario can also affect the term weighting results, since these weights depend on the term frequency of important words). In what follows, we describe in more detail this screencast selection process and the data preparation steps we applied for the two case study projects (WordPress and Firefox).

WordPress Video Selection and Data Preparation:

Common to CMS tools such as WordPress is that they provide a dashboard that allows users to select functionalities and features for creating content pages. Typical steps to create a website using WordPress[18] include a) choosing and purchasing a domain and web hosting, b) installing WordPress, c) choosing and/or installing a WordPress theme and configuring it, d) publishing a page, e) creating a menu, f) configuring the WordPress settings, and, g) installing WordPress plug-ins. Among these steps, publishing a page (or post) and creating a menu are the main steps than do not require installing third-party tools or plug-ins and can be done through the WordPress default dashboard. Therefore, for our study, we therefore selected two scenarios from the WordPress online documentation that describe such dashboard use: 1. "*How to add menus to WordPress*"[19] and 2. "*How to create a post in WordPress*"[20].

For the selection of the screencast scenarios, first we performed different queries on YouTube using keywords being selected from the project's official online documentation[21] (e.g., add new post in WordPress). The first scenario describes an administrative task in which a WordPress user has to enter only a limited amount of free-form text, while for the second scenario, the user has to provide a substantial amount of text. The amount of user-specific text in these scenarios provides us with variation in the data in terms of different signal-to-noise ratios.

We restricted the selection to screencasts to videos that are in HD quality and were uploaded in the same year as each other to determine the most popular version of WordPress (with most videos available on YouTube), since videos that are uploaded in the same year are more likely to be related to the same version of WordPress. Our analysis shows that the largest number of uploads for both scenarios is in 2015. WordPress version 4.3 is the latest version of WordPress in 2015 and therefore its source code will be used in our analysis. While YouTube provides an advanced filtering feature to limit the length of videos to a maximum length (e.g., less than 4 minutes), such filtering would also return videos that are very short or do not contain any useful content, and hence would again add noise in the data set. We therefore eliminated through manual filtering videos that are too short (less than 120 seconds in our data set) and chose videos that were shorter than 10 minutes, since in our data set videos of longer than this would contain more noise or cover multiple topics. After applying the other selection criteria, the length of videos in our data set is between 120 to 480 seconds. Limiting our data to shorter videos provides us with a more balanced data set that contains less noise, since these shorter videos are more likely to only cover one single usage scenario. For our WordPress case study (Moslehi, Adams and Rilling, 2018) we also limited our search to screencasts that have both English narration and contain only English text on the screen. For each scenario, 5 screencasts that met our selection criteria (i.e., HD quality, having a narrator, covering the same WordPress version and having the same upload year) were selected and downloaded using youtube-dl (Table 1).

[18] https://premium.wpmudev.org/blog/a-wordpress-tutorial-for-beginners-create-your-first-site-in-10-steps/
[19] https://www.youtube.com/playlist?list=PLdf7gmFvpFlDU-TLQV83LezrwSP9Pudd8
[20] https://www.youtube.com/playlist?list=PLdf7gmFvpFlCRBsGKdkyVrRbYJRNBAwbS
[21] https://wordpress.com/

Next, we downloaded the source code for WordPress version 4.3 from the project's GitHub repository[22], and manually replicated and recorded the execution traces of these scenarios. We then used Exuberant Ctags to extract source code data (e.g., variables, comments, identifiers - see Section 5.1 for more details). Since most of the WordPress features are developed using PHP, we only considered the PHP files for our study. We extracted for WordPress 554 PHP files, with a median size of 162 lines of code with 25th/75th percentiles of 52 and 511.

Mozilla Firefox Video Selection and Data Preparation:

Firefox is among the most widely used multi-platform Internet browsers on the market. Browsing the Internet involves several remote server calls to asynchronously load remote page elements. Since these remote method calls are not part of the Firefox's core JavaScript or C++ application source code, they are difficult to trace and analyze. We therefore focus in our study only on features that involve modifying or executing browser-level functionality that is executed on a user's local machine (e.g., modifying security settings, managing saved user logins, saving web pages).

Firefox main menu is divided into categories from which we excluded features that are related to design (e.g., "Customize" menu) since they are simple scenarios that typically can be done in few steps. We also excluded plug-ins (e.g., "Add-ons") since they do not exist on the default installation of Firefox and therefore are not directly counted as Firefox features. Instead, we investigated the menu items and their sub-menus to locate features that provide more advanced browser-level features and functionalities such as privacy and security, or preferences and options that are related to web browsing functionalities since they are more popular and complicated.

As a result, for our case study, we selected the following seven scenarios from the Firefox online documentation[23]: 1. "*How to import bookmarks*"[24], 2. "*How to change the default search engine*"[25], 3. "*How to set the homepage*"[26], 4. "*How to save a web page to PDF*"[27], 5. "*How to remove saved logins and passwords*"[28], 6. "*How to clear history and cache*"[29], and 7. "*How to do private browsing in Firefox*" [30]. From these scenarios, we excluded "*How to do private browsing in Firefox*" since the Firefox profiler (i.e., Gecko profiler) does not record execution traces if the browser is in private mode.

Given the long development history of Firefox and the large number of available videos on YouTube covering these Firefox features, we limited our search to screencasts that were uploaded during the last year (at the time of conducting this research) to further improve the ability to locate screencasts that are potentially related to Firefox Quantum (Version 65), which has been available since the beginning of 2019. Similar to the WordPress scenarios, these screencasts contain both information directly related to the Firefox application, but also some user specific (noisy) information (e.g., the scenario "*How to save a web page to PDF*" will typically also show a random web page that will be saved). Like for the WordPress data, we reduced the noise and imbalance in the data by limiting the length of the selected screencasts. For the Firefox videos, after performing an initial manual analysis, we again restricted the video length to 10 minutes since these videos typically only cover a single use case scenario and contain therefore less noise. This, in combination with our other selection criteria (HD quality and upload year), resulted in a video data set with videos that are between 28 to 430 seconds long.

For our Firefox case study, the objective is to evaluate the performance of our approach on a large-scale project, where its backend and frontend implementation of features is not only distributed across different architectural layers but also using different programming languages. To address this challenge, we increased the number of

[22] https://github.com/WordPress/wordpress-develop/tree/4.3
[23] https://support.mozilla.org/en-US/products/firefox
[24] https://www.youtube.com/playlist?list=PLdf7gmFvpFlDcfwNGmefCDCFpolHm_fwy
[25] https://www.youtube.com/playlist?list=PLdf7gmFvpFlC5z8SiLcCXbEzJqBcaGlXS
[26] https://www.youtube.com/playlist?list=PLdf7gmFvpFlD-J_gerUJ1i6URXLGOT2-H
[27] https://www.youtube.com/playlist?list=PLdf7gmFvpFlDzRSj0L1-zZhTzC0KynHhD
[28] https://www.youtube.com/playlist?list=PLdf7gmFvpFlAlxfUyVSyTZyjiPQm-ST7K
[29] https://www.youtube.com/playlist?list=PLdf7gmFvpFlBPWzefVqisCRnBtApQ5yC-
[30] https://www.youtube.com/playlist?list=PLdf7gmFvpFlACZGdcb1Crh6T48VzEfYqT

videos we used as queries for locating source code artifacts to cover these additional scenarios. For this evaluation, videos no longer were required to have an English narrator and only English text on the screen. Instead, we relied on using bigram topic modeling (see Section 5.3) in addition to modifying term frequencies (see Section 5.4) to reduce the effect of noise in our data that may result from having non-English text or characters in the OCR output and errors in the STT output when narrator is not an English speaker. Selected screencasts were then downloaded using youtube-dl (Table 1).

We obtained the Firefox source code used in our study from the project's Mercurial repository[31]. Firefox consists of a large codebase with source code written in different programming languages (e.g., JavaScript, C, C++, Rust, Python, XML, XUL). The Firefox backend is implemented mostly in C/C++, while JavaScript is mostly used for the user interface or frontend development. We performed the experiments on a Mac running OS X, which only uses Firefox code that is developed specifically for OS X platforms. Using again Exuberant Ctags, we extracted the source code facts from JavaScript and C/C++ source code files. The analyzed codebase consisted of 36,666 JavaScript files and 33,440 C/C++ files, with a median size of 45 lines of code for the JavaScript files (25th/75th percentiles of 27 and 83) and a median size of 133 lines of code for C/C++ files (25th/75th percentiles of 64 and 332).

Applying our Methodology: For our case studies, we consider the following screencast data: user actions, text shown in the screencasts and narrator speech (if available). For screencasts with a narrator, we used IBM Watson's speech-to-text service (STT) to automatically transcribe the audio part of these screencasts (Moslehi, Adams and Rilling, 2016). The tool also provides a confidence score that specifies how accurately each term is transcribed from the speech. To reduce the effect of noise and errors in the STT output, in our previous study on WordPress data set we filtered out words from the transcribed text with a low confidence score (below 0.7) (Moslehi, Adams and Rilling, 2018). As mentioned earlier (Section 5.3), in this work we use bigram topic modeling to mitigate the problem of having errors in our STT output by emphasizing the word context. Using bigrams in addition to applying term weighting on speech data will reduce the effect of noise and therefore there was no need to apply a confidence score threshold to the Firefox STT output.

For the processing of the image frames content, we extracted image frames at a rate of **one frame per second** using FFmpeg[32] (Table 1). It should be noted that the number of reported frames (Table 1) corresponds to the frames after manually removing the begin/end of each video, since these parts (e.g., greetings, information related to the video author or YouTube channel, thank-you notes, or closing remarks inviting viewers to like or share the screencast) typically do not contain any information relevant to the scenario.

Table 1 Number of transcribed speech documents and image frames extracted from each video.

Project	Usage Scenario	Number of Screencasts	Number of Speech Documents	Number of Image Frames used for OCR
WordPress	Menu	5	5	1,561
	Post	5	5	749
Mozilla Firefox	Import Bookmarks	12	5	1,349
	Clear History	21	6	1,402
	Set Homepage	21	7	1,393
	Remove Passwords	16	7	1,131
	Save-to-pdf	6	1	680
	Default Search Engine	13	6	694

[31] https://hg.mozilla.org/mozilla-central
[32] https://www.ffmpeg.org/

Next, we apply OCR to extract the textual content of each sampled video frame. We first explored the Tesseract tool[33] used by Ponzanelli et al. in (Ponzanelli *et al.*, 2016). However, the tool requires a significant amount of pre-processing to improve first the quality of the image frames, by removing the image background, resizing the image and removing frame areas that do not contain any text. We instead opted for Google Vision API's Text Recognition service[34], which does not require these preprocessing steps and can also recognize text on images with any background, while providing the (x, y) coordinates of the area in which the text appears on the image frame as well as a confidence score for the recognized words.

Fig. 9 A sample user action in a screencast.

Google Vision API is an image analysis service that is built on top of machine learning models that are trained on a large volume of image data with various characteristics (e.g., image background, quality, text font, language). For evaluating the performance of our user action detection approach (Formula 1, Figure 9) using Google Vision API, we calculated the precision and recall of the user actions that are identified using our approach. We manually verified, using one randomly selected video, the user actions leading to a change of the screen content (e.g., clicks on a button or menu items, scroll up/downs, zoom in/outs) and the corresponding text changes to identify the true positives and false positives.

From our analysis of the raw OCRed text (without any preprocessing), we observed that our approach is able to detect all user actions and text fragments (recall 100%) with a precision of 49%. This low precision is due to errors in the OCRed text of image frames that resulted in 42 out of 121 image frames (35%) being wrongly flagged as user actions. These false positives mainly occur if two subsequent image frames while very similar, contain erroneous OCRed words that result in wrongly identifying a new user action due to partial word differences. However, when we compare the manually extracted user actions to the ones extracted by our proposed approach, the length of the text (i.e., number of words) in the OCRed text that is identified incorrectly by our approach as a user action is short (Median = 4 words). Such false positives (Table 2) can be removed by applying additional preprocessing on the text (i.e., modifying term frequencies and bigram topic modeling). In addition, since for each screencast we merge the text of all identified user actions in one large data set, these relatively few short text outliers will become less important. Table 2 provides results of our statistical analysis of the word length in each OCRed text based on the text differencing approach introduced in formula 1.

In addition, by using our frame sampling approach (with one frame per second), we can reduce the processing cost for image and OCR processing significantly compared to an approach that would process the full frame rate of videos (e.g., 30 frames per second). Also, our approach is scalable since the OCR (text detection) process is executed in a cloud environment, which allow us to take advantage of elasticity and scalability provided by cloud computing.

[33] https://github.com/tesseract-ocr/tesseract
[34] https://cloud.google.com/vision/

Table 2 Statistical analysis of the OCR differencing approach on a randomly selected video.

Observation	Minimum Length	Maximum Length	Mean	Median
True Positive	1	72	19.42	17.4
False Positive	1	58	6.0	4.0

For our topic modeling, we used MALLET[35] to perform LDA. To determine the optimal number of topics for our study, we split the source code data set into a 90% training and a 10% test portion. For the WordPress and Firefox data sets, we used bigram topic modeling (see Section 5.3 and RQ0). To determine the number of topics, we evaluated both models based on their log likelihood values (Figure 8) to identify the "knee" point K where one gets diminishing returns in log likelihood as the number of topics increases. For WordPress bigram topics we identified $K = 55$, for WordPress unigram topics $K = 80$, and for Firefox (bigram topics) $K = 130$, which we used later for our case studies.

Baselines: We evaluate our ranking results against baselines that we manually created by executing the locally compiled and built versions of the projects and recording their execution traces for the scenarios described in the screencasts.

WordPress Baseline Creation: Using Xdebug[36], which is a popular PHP profiler, we create execution traces in the form of a call-graph for each usage scenario of WordPress. We then parsed the call-graph trace to extract the path for each PHP file whose method executions was recorded.

We create two different types of baselines. The first baseline, the *Unique* baseline, contains only files whose methods were executed and that are unique to a given scenario. This baseline can be considered to be more relevant to the technical implementation of the essential steps of a particular scenario while still containing a low amount of generic information. In contrast, the *All* baseline consists of the *Unique* baseline and in addition also includes methods which might be shared with other scenarios. For example, in WordPress functions.php[37] includes methods that are called during the execution of different WordPress features, since these methods are responsible on how a site is publicly displayed. The *All* baseline is more generic, since it includes both some general executions and executions specific to a given scenario.

Firefox Baseline Creation: The execution traces for the Firefox usage scenarios are created using its built-in Gecko profiler[38]. The input to the profiler is created by replicating and recording the execution traces for the scenarios shown by the screencasts. The Gecko profiler is a sampling profiler that interrupts the threads that it is profiling at regular intervals (e.g., 1-2 milliseconds by default depending on the platform) and captures the call stack each time the thread is interrupted. As a result, some calls might be omitted in the call-graph since the profiler does not include them in the snapshots of that execution. To mitigate this threat, we profiled the same scenario several times, creating different trace snapshots, which we then combine in a single trace.

The output of the Gecko profiler contains the fully qualified names of the executed methods for C/C++ or Chrome[39] URLs[40] that are used to reference the JavaScript source code files. The mapping of the Chrome URLs to concrete file names is based on manifest files that contain the rules for resolving the URL into their concrete file paths. Instead of decoding the complex rules to translate Chrome URLs to concrete files we used a simpler approach that uses the JavaScript file name in the URL, method name, and the method's line number, which are provided by the profiler, to locate the corresponding file. Also, in Firefox many of the recorded execution traces

[35] http://mallet.cs.umass.edu/
[36] https://xdebug.org/
[37] https://codex.wordpress.org/Functions_File_Explained
[38] https://perf-html.io/
[39] https://developer.mozilla.org/en-US/docs/Glossary/Chrome
[40] https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/The_Chrome_URL#The_Chrome_URL

are lower-level system calls that are part of the general Firefox initialization. These system calls introduce additional noise in the trace data in terms of being not directly related to the specific scenario. To mitigate this issue, we recorded the execution trace of Firefox while it was in idle mode and subtracted this recorded (idle) trace from that of each scenario.

Evaluation Measures: Since the objective of our feature location approach is to identify a starting point (seed) in the source code to be used for further manual feature exploration, we use Average Precision (AP) and Reciprocal Rank (RR) as evaluation measures to assess the ability of our approach in retrieving true positives at the top of the result set (Keivanloo, 2013):

$$AP = \frac{1}{|R|} \sum_{k \in R}^{n} Precision\ at\ k \qquad (3)$$

$$RR = \frac{1}{rank\ of\ the\ first\ tp} \qquad (4)$$

, with R being the set of all relevant retrieved items and k being the number of retrieved items. We prefer AP over "Precision at k" measure since unlike "Precision at k", AP is generalizable over multiple queries (values of k which in our case are screencast data). Using AP, one can calculate now the mean of the "Precision at k" value for all queries (here we used the median value to avoid bias caused by outlier values). Furthermore, AP considers the position of the true positives in the ranking and is a measure that does not require creating relevancy scores. Also, similar to "Precision at k", AP has this limitation that to be able to provide a fair assessment, using for example "Precision at 1000", the corpus for all executed queries must contain at least 1000 (i.e., k) relevant items (Keivanloo, Roy and Rilling, 2014)

RR can be considered as a complementary measure to AP for assessing the quality of query results. RR considers the position of the first true positive in the search results and is best applied when one has very few true positives (such as in feature location), some of which are located in the top k of the search results. RR therefore evaluates whether the most relevant hits appear at the top of the result set, while AP determines if an approach can retrieve the relevant results and rank them among top hits.

For Firefox, we used directories as our granularity level for the search results, since our initial evaluation at both file and directory granularity levels showed that our approach performs better in locating source code at directory level compared to the file level (our evaluation data can be found online for file[41] and directory[42] granularity levels). Our analysis for locating and ranking source code relevant to the backend development of Firefox at the file granularity level resulted in median values close to zero for both AP and RR, making our approach not applicable at this granularity level. Closer analysis showed that this is due to the large difference in vocabulary between GUI-related features and C/C++ source code.

Table 3 Mean, median, 25th Percentile, 75th Percentile, maximum, and minimum number of JavaScript and C/C++ files.

Language	Mean	25th Percentile	75th Percentile	Median	Max	Min	#Directories
C/C++	14.29	1	10	3	2014	1	2623
JavaScript	14.91	1	10	3	1660	1	4268

However, re-applying our approach at directory level granularity, we were able to locate and rank in most cases the directories that contain the relevant files. While this approach no longer directly locates relevant individual files, it still allows for a reduction of the search space and provides users with an approach to partially automate the feature location process. Using the directory level granularity, a user only has to search manually through an individual (relevant) directory for the source code file(s) implementing the particular feature rather than manually searching across all directories. These ranked directories of files can therefore be used as a starting point (seed)

[41] https://mcislab.github.io/publications/2020/emse_parisa/Firefox-File-Level-Analysis.zip
[42] https://mcislab.github.io/publications/2020/emse_parisa/Firefox-Directory-Level-Analysis.zip

for further source code analysis during the feature location process. Table 3 presents statistics about the number of JavaScript/C/C++ files in directories of Firefox source code containing such files.

Table 4 Random guess probability of the baseline files out of the total corpus of 554 source code files in WordPress, 36,666 JavaScript files, 33,440 C/C++ files, and 7,433 directories containing JavaScript or C/C++ files in Firefox.

Project	use-case Scenario	Granularity	Baseline	#files or #dirs	Random Guess %
WordPress	Post	file	All files	92	17
			Unique files	14	3
	Menu		All files	92	17
			Unique files	13	2
Firefox	Import Bookmarks	directory	C/C++	56	0.7
			JavaScript	2315	31
		file	C/C++	156	0.5
			JavaScript	1309	4
	Clear History	directory	C/C++	215	3
			JavaScript	1294	17
		file	C/C++	1298	4
			JavaScript	1420	4
	Set Homepage	directory	C/C++	27	0.3
			JavaScript	1288	17
		file	C/C++	66	0.2
			JavaScript	1447	4
	Remove Passwords	directory	C/C++	262	3
			JavaScript	1299	17
		file	C/C++	1898	6
			JavaScript	1470	4
	Save-to-pdf	directory	C/C++	25	0.3
			JavaScript	1131	15
		file	C/C++	63	0.2
			JavaScript	1183	3
	Default Search Engine	directory	C/C++	30	0.4
			JavaScript	1331	18
		file	C/C++	91	0.3
			JavaScript	1573	4

We also calculated random guess probabilities for both WordPress and Firefox, which we use as a baseline for our result comparison. The random guess probabilities were calculated by dividing the number of files/directories in the baseline by the number of all files/directories in the project. For the random guess probabilities, we calculate how probable it is that a developer by randomly (without considering any bias or prior knowledge) selecting files/directories, would select all the files/directories that are captured in the scenario baseline (Table 4).

For the evaluation, we use the top 10 hits (files) in WordPress and top 100 hits (directories) in Mozilla Firefox. Our initial analysis showed that our approach would only yield no or very few true positives in the top 10 results using the chosen granularity levels. Instead, we used top X/100 as an evaluation measure for the Mozilla Firefox data set, since Firefox is not only a significantly larger project than WordPress, and its front/backend implementation are in distant architectural layers (compared to the WordPress implementation), but it also relies on different programming languages. Furthermore, since the maximum number of directories in our baseline for Firefox may contain fewer than 100 directories in the complete result sets for the baselines, we therefore consider top X/100 (Keivanloo, 2013), with X<=100 and X being determined by the number of directories in the baselines that we extracted for Firefox. For example, if the baseline contains only 60 directories, then the top 60 results are evaluated.

6.2. Case Study Results

RQ0. What is the performance of our approach using unigram topic modeling vs. bigram topic modeling?

As we use LDA in our linking methodology, different factors can affect its performance, including the number of topics (see Section 5.3) and the gram size. In our previous work (Moslehi, Adams and Rilling, 2018), we applied unigram topic modeling on our WordPress data set (Figures 10 and 11). In this paper, we extend our previous analysis using unigram topic modeling to also include bigrams (Figures 12 and 13) and compare the performance of both gram sizes. As part of bigram topic modeling, the number of topics was reduced from K = 80 (unigram) to K = 55 (bigram), with few topics needed to form a particular well-defined topic context.

Results for "menu" Scenario: Figure 10 shows that using unigram topic modeling for the *All* baseline, except for the speech data, the RR values are always 100%, meaning that the first true positive is ranked at the top of the result set. Using bigram topic modeling (Figure 12), we have the same results and the median RR value of speech data not only is improved by 75% but also the variance in the distribution of the RR values is reduced. In case of the *Unique* baseline, the RR value of the speech data improved by 30%. This is while the median RR values when using GUI or GUI-and-speech data decreased to 50% and the variance when using GUI and speech data is removed.

The median AP values of bigram topic modeling when using the *All* baseline are all improved compared to unigram topic modeling. Especially when using speech data, the median AP improves by 70% and the variance in the boxplot is reduced. Evaluations against the *Unique* baseline shows that for the speech data the median AP values improved by 30%, while the variance in other boxplots is decreased and their median AP values are also decreased on average by 19% using bigram topic modeling.

Results for "post" Scenario: The evaluations against the *All* baseline (Figures 11 and 13) shows that the rank of the first true positive drops from the first to the second hit when using GUI weighted or GUI-and-speech weighted data in bigram topic modeling. However, using bigram topic modeling improves the median RR of speech data by 89% and reduces the variations in the box plots. In case of the *Unique* baseline, although the median RR values decreased for five out of six data types using bigram topic modeling, this value increased for the speech data from 0 to 12% and the variations are either removed or reduced which makes the results more reliable.

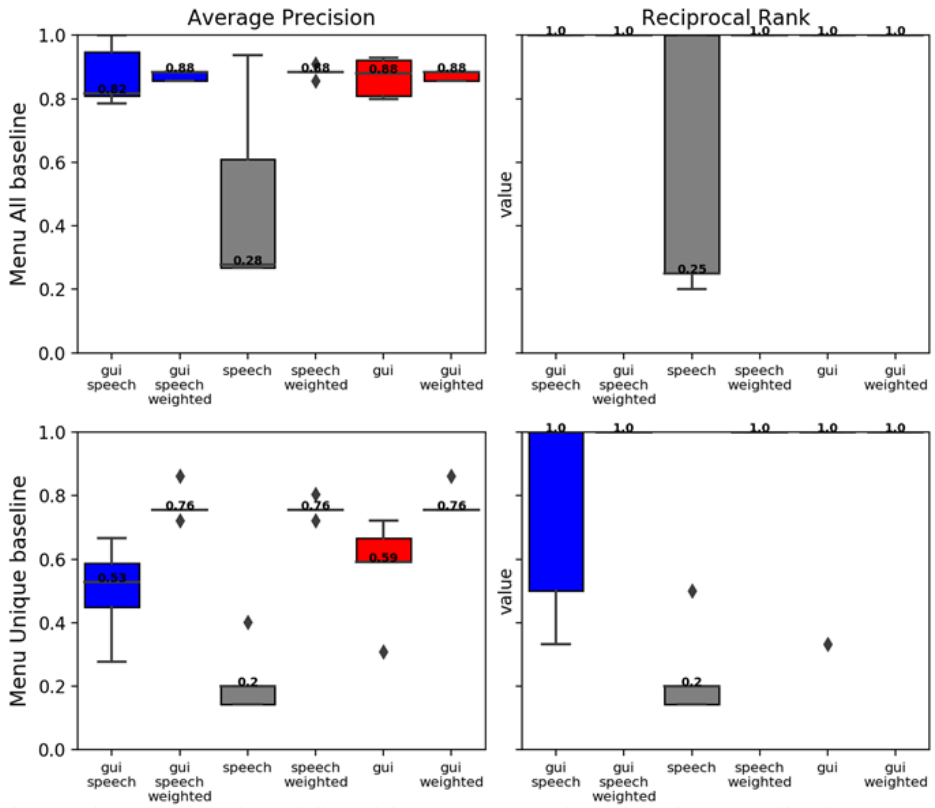


Fig. 10 RR and AP boxplots for the unigram topic modeling of the "Menu" scenario evaluated against All and Unique execution trace files, using both GUI/speech data, only GUI or speech, either with weighted or unweighted metrics.

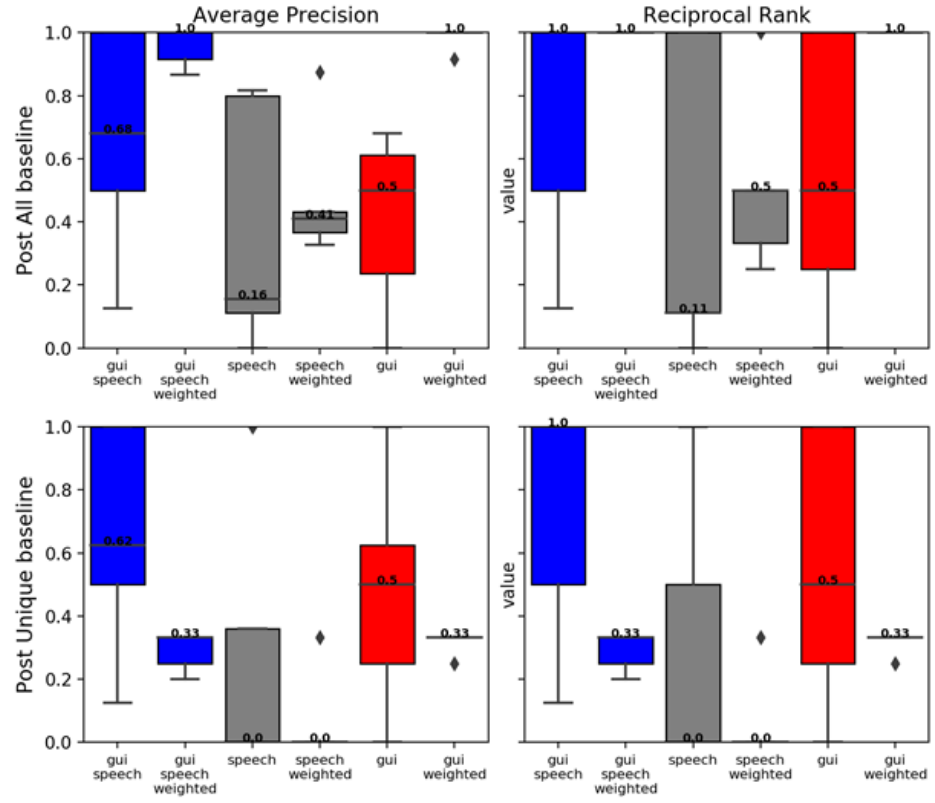


Fig. 11 RR and AP boxplots for the unigram topic modeling of the "Post" scenario evaluated against All and Unique execution trace files, using both GUI/speech data only, GUI or speech, either with weighted or unweighted metrics.

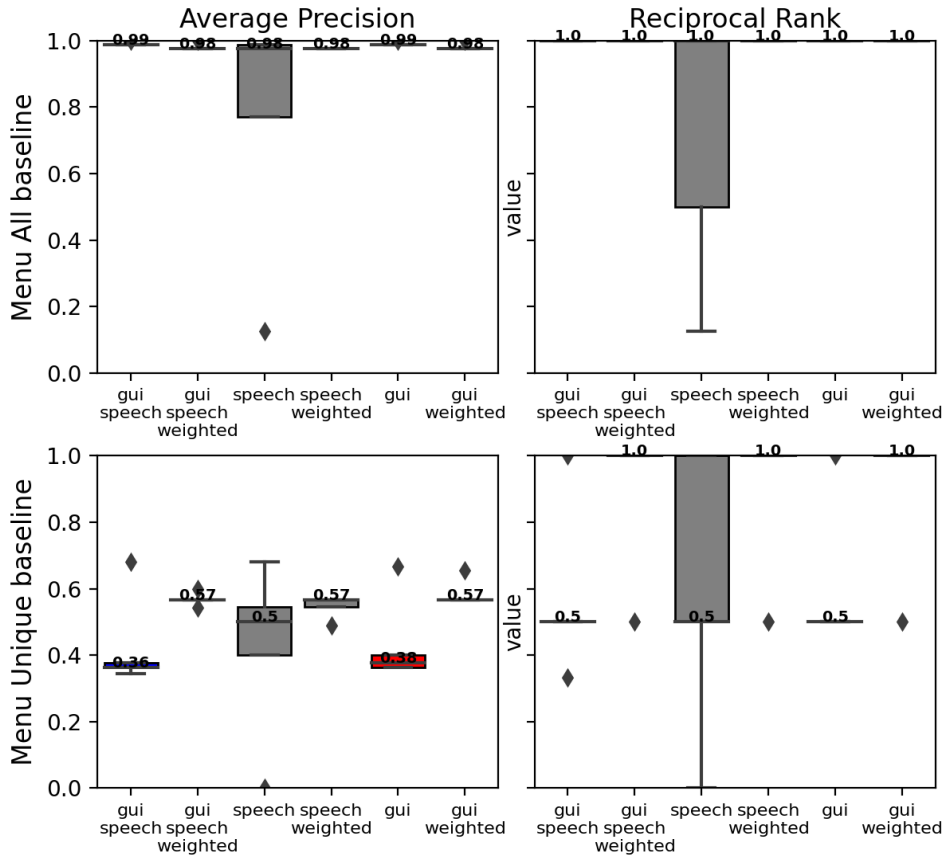


Fig. 12 RR and AP boxplots for the bigram topic modeling of the "Menu" scenario evaluated against All and Unique execution trace files, using both GUI/speech data, only GUI or speech, either with weighted or unweighted metrics.

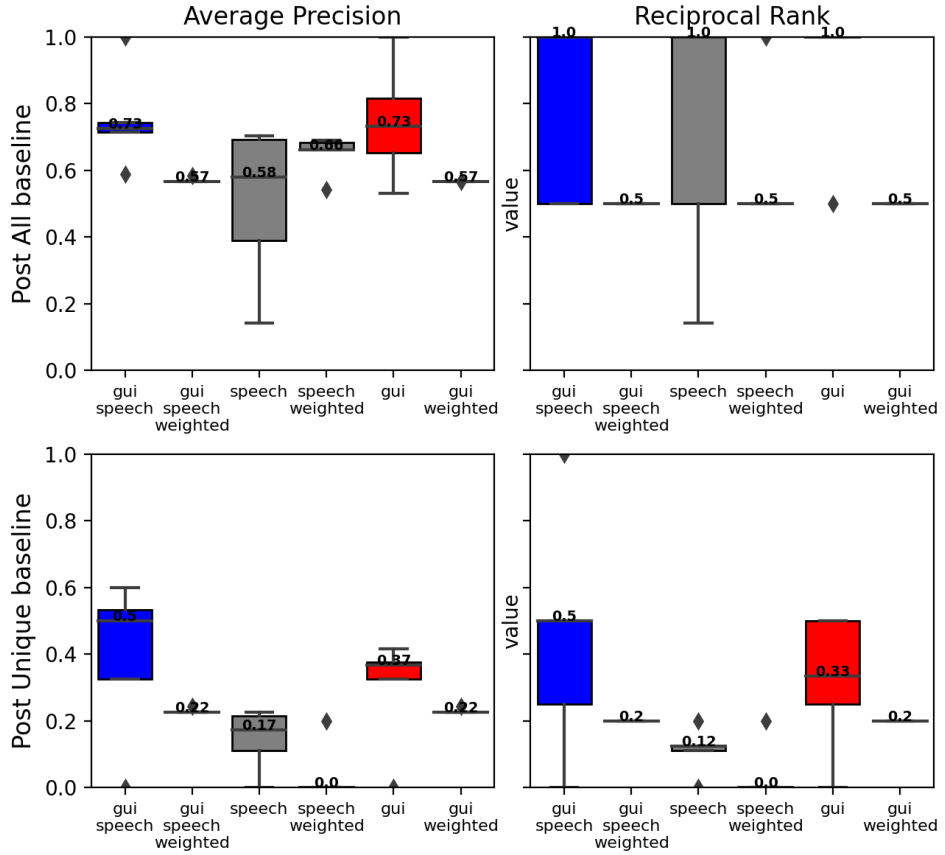


Fig. 13 RR and AP boxplots for the bigram topic modeling of the "Post" scenario evaluated against All and Unique execution trace files, using both GUI/speech data only, GUI or speech, either with weighted or unweighted metrics.

Evaluations against the *All* baseline using bigram topic modeling (Figure 13) show that the AP values of the GUI weighted, and GUI-and-speech weighted dropped from 100% (using unigram topic modeling) to 57%. This means that still the number of true positives is higher than the number of false positives in the top 10 results, and the RR values show that the first true positives appear at the second rank of the top hits. For the other data types the variance in the box plots reduced and the median AP values increased especially for the speech data the increase is by 42%. The evaluations against the *Unique* baseline show that the median AP values for all data types except for the speech data dropped on average by 9.4% while the variance in the boxplots is reduced and the median AP of the speech data increased from 0 to 17%.

Conclusion: While the improvement in median AP and RR values varies between using bigram vs. unigram topic modeling, in general, using bigram topic modeling reduced the variance in the boxplots, which makes the results more reliable. Also, in all baselines, and for both AP and RR, bigram topic modeling improved the performance of the approach when using speech data, which compared to GUI data is shorter. As a result, we use bigram topic modeling in the rest of the paper.

RQ1. How accurately can source code files be located from screencasts?

In what follows, we report on the AP and RR results that we obtained when we take advantage of both GUI and speech data for the linking of screencast content to source code (Figures 12, 13, 15, and 16 the far-left blue boxplots).

Results for WordPress:

Results for "menu" Scenario: Figure 12 shows that the median AP for the *All* baseline is 0.99. Having an AP above 0.5 indicates that we have more true positives than false positives in the top 10 hits, which can be considered a good result for average precision. Given that there are among the 554 WordPress files only a total of 13 PHP files (true positives) for the *Unique* baseline, while the median RR value is 0.5, meaning that the first true positives appears at the second top of the result set, the median AP is 0.36 which means that every third item in the result set is a true positive.

More importantly, all screencasts had an RR value of 100% for the *All* baseline, and 100% of the RR values for the *Unique* baseline were 50%. With the first true positive being ranked first or second indicates that our approach can provide useful seeds for further feature location.

Results for "post" Scenario: Figure 13 shows that the median of RR for the *All* baseline is 100% and for the *Unique* baseline this value is 50%, which indicates that even for a noisier data set (e.g., "post" scenario), our approach is able to rank in most cases the relevant result in first or second position. Also, the median of AP is 0.73 for the *All* baseline and 0.5 for the *Unique* baseline, which shows that, in most cases, the number of true positives that our approach will return is more than (*All* baseline) or equal to (*Unique* baseline) the number of false positives in the top 10 hits.

Further evaluation of both baselines (Table 4) shows that our approach always outperforms random guessing by a factor of at least 16.66 for the *Unique* baselines and 4.29 for the *All* baselines. Random guessing refers here to the process where one would try to correctly guess all relevant files in each baseline.

Our analysis also shows that the *Unique* baseline contains fewer files and therefore fewer true positives that can be ranked compared to the *All* baseline. Also, for the *All* baseline data set, files such as post.php that are commonly ranked at the top of the result set, contain methods that are directly related to features (i.e., GUI items), and therefore have a higher similar term frequency with the screencast artifacts (GUI and speech text).

Results for Mozilla Firefox:

We present the detailed AP and RR results for the two scenarios of "*Import Bookmarks*" and "*Default Search Engine*" where the latter belongs to the group of scenarios with more noise in the data (i.e., "*Set Homepage*", "*Save-to-pdf*", "*Default Search Engine*"). The results are shown again in the far-left blue box plot of Figures 15

and 16. Also, a summary of the median AP and RR values for all scenarios is shown in Figure 14. The complete set of all box plots for the Mozilla Firefox scenarios is available online[43].

Results for "Import Bookmarks" Scenario: Figure 15 shows that the median of AP is 0.18 for the C/C++ files and 0.23 for the JavaScript files. This reflects that every fifth result is a true positive among the top hits. The RR values for both C/C++ and JavaScript are the same, with 0.2. For the JavaScript results 25% of the screencasts have an RR value of more than 0.5, which means that the first true positive appears at rank 1 or 2.

Results for "Default Search Engine" Scenario: Figure 16 shows that also for this scenario JavaScript results outperform our C/C++ results. The first true positive for the JavaScript files appears at rank 5 and for the C/C++ files the first true positive appears at rank 7 of the top hits.

Our study (Figure 14) shows that, scenarios that are more likely to contain noise or irrelevant data in their GUI text (i.e., "*Save-to-pdf*", "*Default Search Engine*", and "*Set Homepage*") have the lowest median AP and RR values for the C/C++ evaluations. We also observed that noise in the data has a negative effect on both JavaScript and C/C++, however the RR values for the JavaScript (frontend) are less affected than the RR values for the C/C++ files.

In our study, the median AP for both JavaScript and C/C++ always outperform random guessing (Table 4), except for the random guess value of the JavaScript directories of the "*Import Bookmarks*". This is because the speech data in this scenario is of very low quality, due to errors in the output of the STT tool (caused by the accent of the narrators) and the significant amount of noise in the data, since the narrators not only talk about importing bookmarks from Firefox but also how to export bookmarks from another browser. As a result, after preprocessing the speech text these documents become very short which negatively affects the LDA performance.

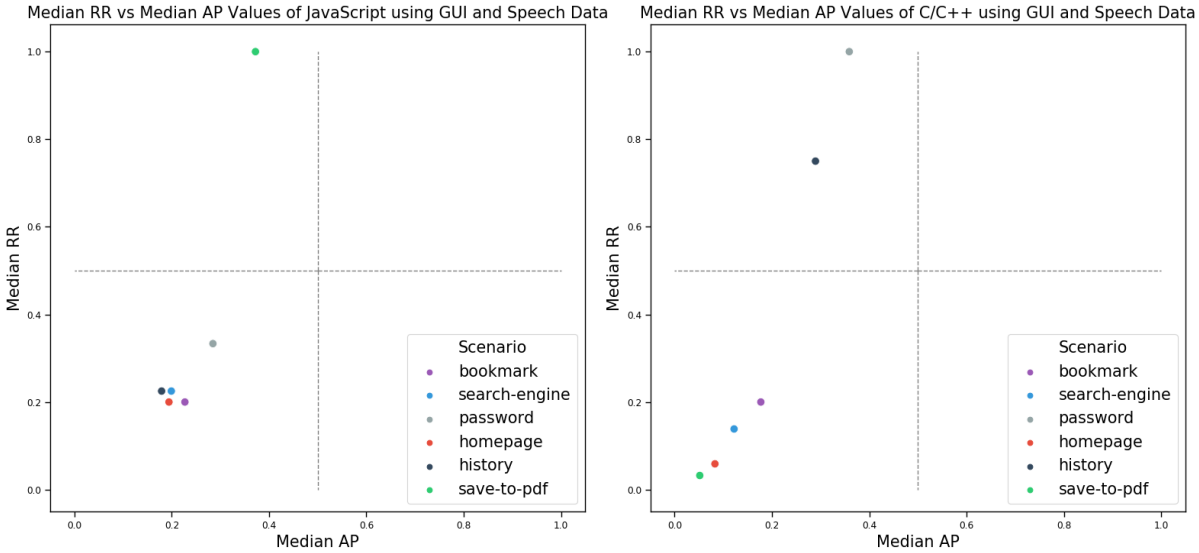


Fig. 14 Scatter plot of the median AP and RR values for each scenario and each language in Mozilla Firefox using GUI and Speech Data.

Conclusion: In WordPress, our approach can successfully rank the first relevant item at the first and second rank of the result set when using the GUI and speech data set, with the results for the "*menu*" scenario being more precise since there is less noise in this data set, which leads to having more terms in common with the source code data set. Even for noisier scenarios, our approach outperforms random guessing (Table 4), by being able to retrieve 50% or more of the true positives in the top 10 results. In Mozilla Firefox, with its larger codebase, it is more challenging to obtain accurate results. However, the median AP and RR values of all scenarios show that the approach can return the first true positives in rank 1 to 30 (in the worst case) of the top *X*/100 results. Also, except for the median AP of the JavaScript results of the "*Import Bookmark*" scenario, in all other scenarios, the approach outperforms random guessing.

[43] https://mcislab.github.io/publications/2020/emse_parisa/Firefox-Directory-Level-Analysis.zip

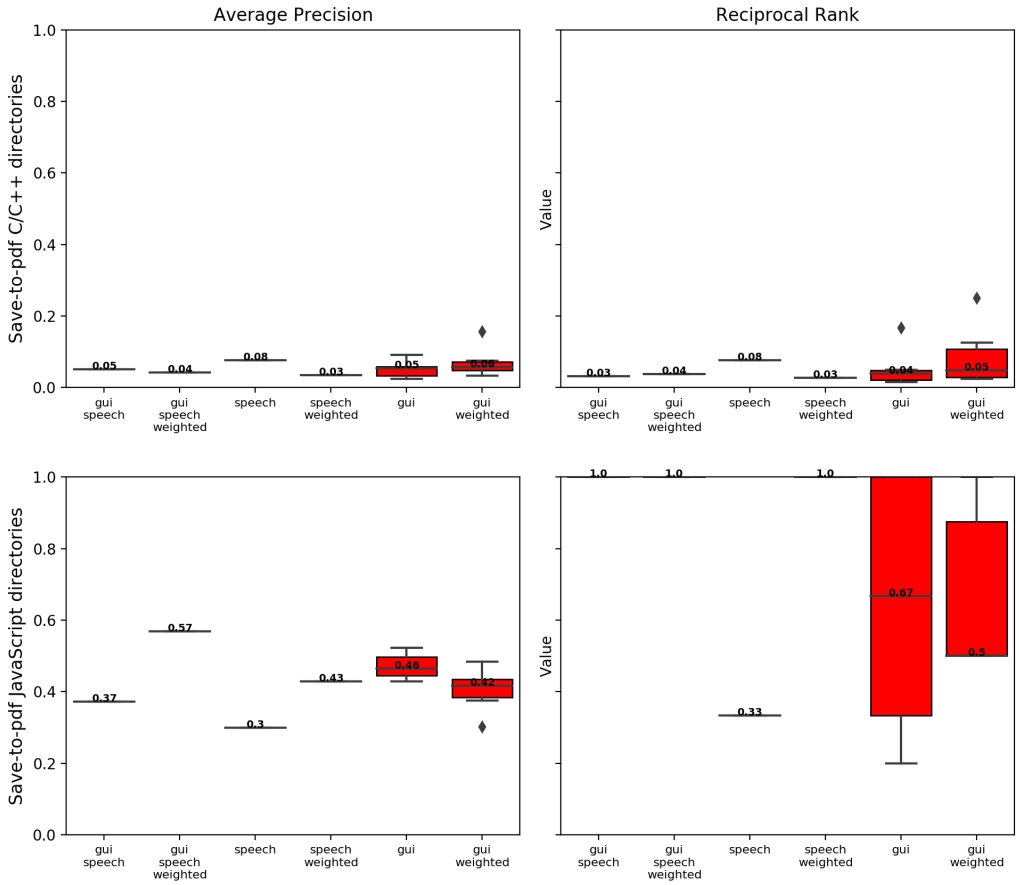


Fig. 15 Boxplots of RR and AP for the "Default Search Engine" scenario evaluated against C/C++ and JavaScript files, using both GUI/speech data, only GUI or speech either with weighted or original metrics.

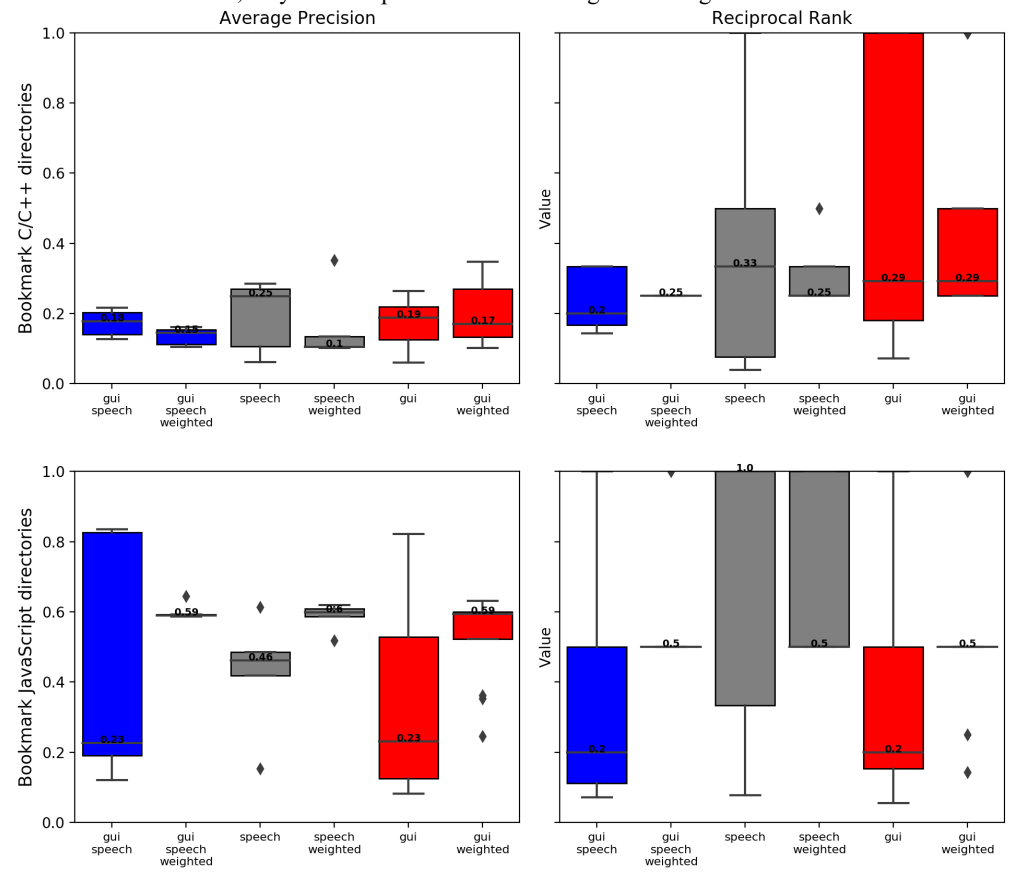


Fig. 16 Boxplots of RR and AP for the "Import Bookmarks" scenario evaluated against C/C++ and JavaScript execution trace files, using both GUI/speech data, only GUI or speech either with weighted or original metrics.

RQ2. How does applying our term weighting approach on GUI and/or speech data affect the feature location performance?

For this research question, we analyze and compare non-weighted and weighted results (boxplots) in Figures 12, 13, 15 and 16 ("gui speech" and "gui speech weighted", "speech" and "speech weighted", "gui" and "gui weighted") to gain insights on how adjusting the term frequencies in the GUI and/or speech data (Section 5.3) affects our linking approach.

Results for WordPress:

Results for "menu" Scenario: The blue boxplots ("gui speech" and "gui speech weighted") in Figure 12 show that, with the adjusted term frequencies, the median RR for both *All* and *Unique* baselines is always 100%, with our approach ranking the relevant results at the top of the result set. For the *Unique* baseline, the RR value improves from 50% when using "gui speech" data to 100% after applying term frequency rebalancing, showing that rebalancing the combination of GUI and speech data improves the RR for this baseline.

When comparing the AP values (non-weighted vs. weighted) of the *All* baseline we see that in this case applying term weighting does not noticeably change the performance of our approach. However, for the *Unique* baseline, term weighting improves the median AP by around 21%.

For the "speech" and "speech weighted" data sets in Figure12, both the *All* and *Unique* baselines applying term weighting on the speech data reduce the variance in the AP and RR results and improved the median AP and RR values, by 7% and 50% respectively, when evaluated against the *Unique* baseline.

For the "gui" and "gui weighted" data sets (red box plots in Figure 12), the RR values for the *Unique* baseline improved by 50% after applying term weighting. While the median AP value for the *All* baseline does not noticeably change, the median AP for the *Unique* baseline after applying term weighting improves by 21%, reflected by the reduction in the variation in the boxplot.

***Results for "post" Scenario*:** The blue boxplots ("gui speech" and "gui speech weighted") of both *All* and *Unique* baseline (Figure 13) show that, modifying the term frequencies reduces the median AP and RR values while reducing the variations which makes the results more precise.

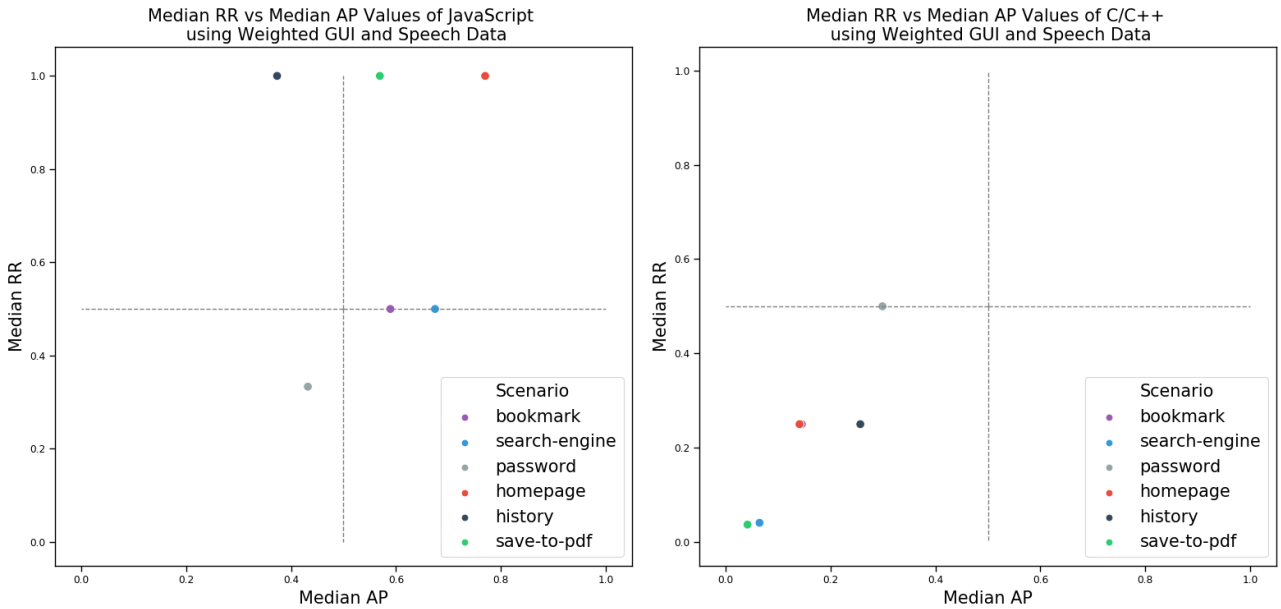


Fig. 17 Scatter plot of the median AP and RR values for each scenario and each language in Mozilla Firefox using Weighted GUI and Speech data.

Comparing the gray boxplots ("speech" and "speech weighted") in Figure 13 shows that, for the *All* baseline applying term weighting improves the median AP and reduces the variation in the plots. The median RR value

shows that after applying term weighting the rank of the first true positive drops to the second hit while the results are more precise with no variation in the plots. For the *Unique* baseline the median AP and RR values drop to 0 after applying term weighting.

Analyzing the red boxplots ("gui" and "gui weighted") in Figure 13 shows that, for the *All* baseline the median AP value dropped by 16% and the median RR value dropped from 100% to 50% with still having the first true positive ranked at the second hit of the result set. The variations of the boxplots are also removed by applying term weighting making the results more precise. At the same time, for the *Unique* baseline, although the variations are again removed by applying term weighting, the median values are not improved and had an 15% and 12% decrease in both AP and RR values respectively.

A more detailed analysis of these somewhat unexpected decreases in both AP and RR values in the results shows that this decrease is due to the weighted documents being shorter after the removal of less important words (noise) with a low corpus-wide frequency, resulting in a lower TF_j (the occurrences of words).

Results for Mozilla Firefox:

Results for "Import Bookmarks" Scenario: Our evaluations of the blue box plots ("gui speech" and "gui speech weighted") for C/C++ file directories (Figure 15) show that after adjusting the term frequencies, the median AP slightly decreases, while the median RR increases by 25% (from rank 5 to rank 4). Rebalancing of the data reduces the variance in the RR results and the improvement from the rebalancing was significantly larger for the JavaScript compared to the C/C++ results. Also, the variances in both AP and RR results have significantly be reduced and the first true positive's rank improves from rank 5 to rank 2. The median AP of 0.59 and median RR of 0.5 means that while the first true positive is ranked at rank 2 of the results, every 3rd item on the result set is a false positive.

Analyzing the "speech" and "speech weighted" boxplots in Figure 15 shows that, for C/C++ directories, adjusting term frequencies reduces the variance in both the AP and RR values and reduces the median values for both boxplots on average by ~1%. For the JavaScript results, term weighting reduces AP and RR variations in the boxplots and increases the median AP value by 14% and reduces the median RR value by 50%. However, the first true positive is still ranked first or second in 50% of the time.

Comparing the "gui" and "gui weighted" (red boxplots) results in Figure 15 shows that, for the C/C++ directories, applying term weighting results in a small decrease (2%) in the median AP value with no change in the median RR value. At the same time, the variation in the boxplots is reduced only for the RR values and marginal changes are observed for AP values. The effect of applying term frequency rebalancing for the JavaScript results shows that the median AP and RR values improved on average by 33% and variations in the boxplots are reduced (AP) or removed (RR).

Results for "Default Search Engine" Scenario: After rebalancing the "gui speech" data (blue box plots in Figure 16), both median AP and RR increase. The median AP increases from 0.2 to 0.67 and the median RR increases from 0.22 to 0.5 indicating that the first true positive being ranked now 2 in the top hits of JavaScript evaluations. In contrast, for the C/C++ directories, rebalancing did not improve our ranking results, which is reflected by lower median AP and RR values.

Rebalancing the "speech" data (gray boxplots in Figure 16) shows that, for the JavaScript results the median AP and RR values improved by 13% and 67% respectively. While for the C/C++ results, rebalancing the speech data results decreased the median AP and RR values by 5%.

Applying term weighting on the "gui" data (red boxplots in Figure 16) for the C/C++ directories, slightly changes the median AP and RR values and the variations (decrease in AP and increase in RR). For the JavaScript directories, the median AP and RR values decreased by 4% and 17% respectively and the variation in the boxplots for the RR values was also reduced.

Based on the median values for all scenarios, term frequency rebalancing improves both AP and RR values for most of the JavaScript results, except for the median AP and RR values of "gui weighted" and the RR value of "speech weighted". For the C/C++ directories, term weighting did not show any improvements for the AP and RR values. Also, similar to the WordPress "*Unique*" baseline evaluations, C/C++ documents contain a low number of words that can be linked to screencast documents' words. Therefore, removing noise from screencasts by rebalancing results in shorter documents, which limits the applicability of LDA and leads to a lower median RR and AP values for the C/C++ evaluations. This contrasts with the frontend development in JavaScript, which handles GUI events and therefore shares more high-level words (i.e., GUI related words) with the GUI of the Firefox browser and the adjustment of term frequencies improved the overall results.

***Conclusion*:** Our WordPress analysis shows that, for scenarios with less noisy data (e.g., menu.php), applying a term weighting approach can improve the AP and RR results for locating project features in the source code by removing variations. However, for a scenario with noisy data, applying such term weighting often results in lower AP and RR values. Similarly, in Mozilla Firefox, term frequency adjustment improves results when the source code documents contain more high-level concepts (e.g., containing direct references to GUI elements).

RQ3. Are both speech and GUI text data required for feature location?

In what follows, we compare the impact of using either GUI (red boxplots) or speech (gray) screencast data or a combination of both (blue) on the performance of our feature location approach (Figures 12, 13 and Figures 15, 16).

Results for WordPress:

Results for "menu" Scenario:

Using speech data only: The gray boxplots in Figure 12 show the feature location results our approach achieved for the WordPress *All* and *Unique* baselines. For the *All* baseline, using speech data only, we obtained a median AP of 98% and a median RR 100%. For the *Unique* baseline, both median AP and RR are 50%. Rebalancing the term frequencies shows an improvement in the variations of the AP and RR values for both baselines and improvements for the median AP and RR of the *Unique* baseline. These improvements through the rebalancing are due to the fact that the unbalanced speech data contains a significant amount of noise, which introduces both many false positives and false negatives.

Using GUI data only: For the *All* baseline the first true positive is ranked at the top (median RR = 100%) and in case of the *Unique* baseline the first true positive is ranked at the second top result. The median AP value for the *All* baseline is 0.99 and 0.38 for the *Unique* baseline. Here also, rebalancing the GUI-only screencast shows improvement in the AP and RR values of the *Unique* baseline.

Results for "post" Scenario:

Using speech data only: As Figure 13 (the gray boxplots) show, using only speech data without rebalancing for the *All* baseline, the median AP and RR values are 58% and 100% which are as performant as other data types. However, for the Unique baseline these values are lower compared to the other data types. Rebalancing the speech data in the *Unique* baseline results in median of 0 for both AP and RR, which is again due to having shorter documents after rebalancing and having fewer common terms with the source code files in the *Unique* baseline.

Using the GUI data only: The median AP and RR for the *All* baseline is 73% and 100%, which can be interpreted as our approach being able to rank the first true positive at the top of the results and the number of true positives being larger than the number of false positives in the top 10 hits, while the observed performance for *Unique* baseline is lower. Rebalancing of the GUI data reduces the performance for both baselines and both measures while removing the variations.

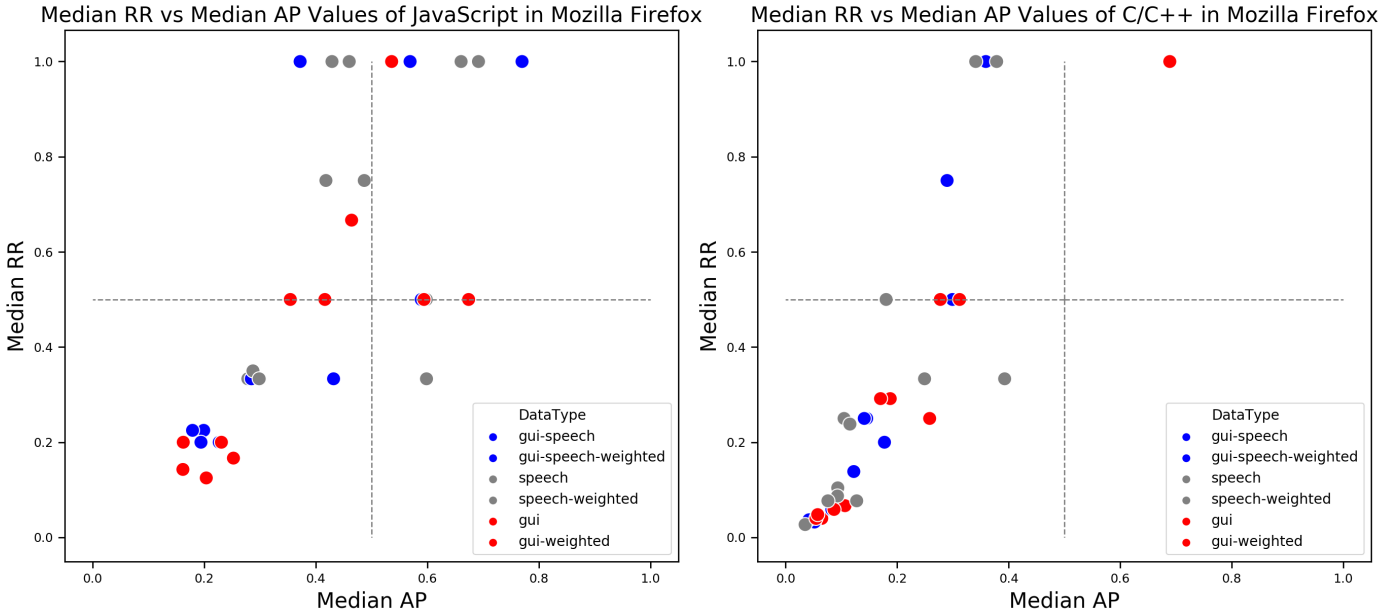


Fig. 18 Scatter plot of the median AP and RR values for each data type and each language in Mozilla Firefox.

Results for Mozilla Firefox:

Using speech data only: For linking the screencast speech data to C/C++ source code (Figures 15 and 16), adjusting term frequencies in speech data always results in lower median AP and RR values compared to the speech-only data, which is not rebalanced. Removing noise from speech-only documents leads to a poorer performance (yet smaller variance in the results) of our approach since we are now dealing with shorter documents that share less words with C/C++ files, affecting the performance of LDA.

Using GUI data only: The "*Import Bookmarks*", "*Clear History*", "*Set Homepage*", and "*Remove Passwords*" scenarios have the largest number of image frames (Table 1) and therefore are expected to have more GUI data. However, Table 5 shows that speech data in general outperforms the GUI data. This is due to the fact that the speech data contains more words that are related to the scenarios compared to the noisier GUI data, which contains also many unrelated terms. Also, in most cases, using speech data or combining it with the GUI data and applying term frequency rebalancing, will lead to improve the AP.

For the JavaScript results specifically, Table 5 and Figure 18 show that using speech-only, rebalanced speech (sw), or rebalanced GUI with speech data (gsw) will result typically in the highest median RR and/or AP values, except for the "*Import Bookmarks*" and "*Remove Password*" scenarios, where the speech-only data produced the highest median RR value.

The "*Save-to-pdf*" scenario has the lowest observed median AP and RR values for C/C++ files, followed by "*Default Search Engine*" and "*Set Homepage*" scenarios, which have also quite low median AP and RR values for their C/C++ evaluations. A common characteristic of these scenarios is that they contain a significant amount of noise in the GUI data, since the narrator opens as part of these scenarios a web page that contains text unrelated to these scenarios. In all cases with noisy GUI data, the speech data becomes the more reliable data source.

In WordPress and Firefox, a combination of GUI data with speech data will in general provide a good performance, while rebalanced speech data in Firefox is overall the most reliable information source.

Table 5 Data sets with the highest median AP and RR values for each project and each baseline (g: gui, s: speech, w: weighted).

Project	Scenario	Baseline	Highest Median AP	Highest Median RR
WordPress	Menu	All files	0.99 (gs, g)	1 (gs, gsw, sw, g, gw)
		Unique files	0.57 (gsw, sw, gw)	1 (gsw, sw, gw)
	Post	All files	0.73 (gs, g)	1 (gs, s, g)
		Unique files	0.5 (gs)	0.5 (gs)
Mozilla Firefox	Import Bookmarks	C/C++ dirs	0.23 (s)	0.33 (s)
		JavaScript dirs	0.6 (sw)	1 (s)
	Clear History	C/C++ dirs	0.39 (sw)	1 (s)
		JavaScript dirs	0.66 (sw)	1 (gsw, sw)
	Set Homepage	C/C++ dirs	0.18 (s)	0.5 (s)
		JavaScript dirs	0.77 (gsw)	1 (gsw, gw)
	Remove Passwords	C/C++ dirs	0.69 (gw)	1 (gs, s, gw)
		JavaScript dirs	0.69 (sw)	1 (s, sw)
	Save-to-pdf	C/C++ dirs	0.08 (s)	0.08 (s)
		JavaScript dirs	0.57 (gsw)	1 (gs, gsw, sw)
	Default Search Engine	C/C++ dirs	0.12 (gs, s)	0.24 (s)
		JavaScript dirs	0.67 (gsw, gw)	0.5 (gsw, gw)

Conclusion: In WordPress, our analysis has shown that for videos with relatively low noise levels, both GUI and speech data by themselves can be sufficient for feature location while they require additional rebalancing. For screencasts with noisier data (e.g., "*post*" scenario), both speech or GUI data (without rebalancing) on their own can be sufficient for locating high-level or less technical source code (i.e., *All* baseline) while for the *Unique* baseline neither of the data types on their own do perform consistently well. In Mozilla Firefox, speech-only data or rebalanced speech data is the most important information source in locating relevant source code artifacts.

RQ4. What is the performance of the approach in locating source code related to frontend and backend of a project?

As mentioned earlier, our screencast data set contains feature descriptions that are typically related to high-level GUI elements. In this research question, we evaluate the ability of our approach to link features shown in screencasts to their corresponding frontend and backend implementation. More specifically, we analyze if our approach can successfully trace features shown in screencasts to their corresponding low-level (i.e., backend) and high-level (i.e., frontend) implementation. Next, we report our analysis results for Mozilla Firefox, in which frontend and backend development of features are not only implemented in different architectural layers but also using different programming languages (i.e., JavaScript and C/C++).

Mozilla Firefox uses C/C++ for its backend development to implement the scenario logic and typically has only very few references to any GUI elements of the Firefox browser. As our analysis shows, the median AP values for mapping screencast content to the C/C++ implementation is rather low, since fewer of these high-level terms or GUI-related elements can be directly found in the backend implementation. For the backend implementation, a combination of GUI and speech data produced always the highest median AP or RR values.

JavaScript is used by Firefox to implement and handle GUI events as part of the frontend development and therefore more GUI-related words appear directly in the JavaScript source code. Our study shows that the median AP and RR for the JavaScript data set are always significantly higher than the ones we obtained for the C/C++ data set.

C***onclusion:*** While our approach can trace feature implementations from screencasts to low-level backend development source code, its accuracy is noticeably higher when applied to the frontend implementation that handles high-level or GUI-related aspects of an application.

RQ5. How accurately can the approach locate source code directories compared to a guided search approach?

To validate the effectiveness of our approach in locating source code directories that are relevant to a tutorial screencast, we compared our Mozilla Firefox evaluation results not only to a random guess approach (Section 6.1, Table 4), but also to a basic **guided search** that is more typically used by developers. In such a guided search (Gray, 2007) approach, a developer (or observer) uses their (programming) knowledge and expertise (i.e., domain knowledge) to search for objects (or source code artifacts that implement a feature) amongst other (distracting) objects. Using this approach, developers narrow their search space more swiftly while locating the source code implementing a feature. For the evaluation of the guided search approach, we again use AP and RR as performance measures.

To simulate this process, we selected the longest screencast for each scenario of our Firefox data set. It should be pointed out that captions of GUI components that are demonstrated in a screencast are not necessarily mentioned or similar to the titles or descriptions of a screencast that are usually used by video portals to search for relevant content. Also, our goal is to simulate a developer who views a screencast and then decides based on the application features demonstrated in the screencast to locate the source code for the viewed features. We therefore manually extracted the text of buttons, labels, and menu items on the GUI that are used (e.g., clicked on) or discussed by the narrator in the screencast while performing an action. Instead of using an IDE or text editor to index the Firefox source code, we used Searchfox[44], a source code indexing tool that is available for Mozilla Firefox, to search for the exact words and patterns that we extracted from the screencasts' GUI widget labels. Searchfox allows for advanced filtering options (e.g., case sensitive matching, regular expression matching, and path filtering) that also help with locating exact words and patterns. In what follows we describe how we simulated a developer's actions who tries to locate the source code for the viewed features on a screencast:

Since we are interested in the frontend and backend feature implementations of Firefox that are developed in JavaScript (i.e., .js) and C/C++ (i.e., .h, .cc, .c, .cpp) source code directories respectively, we limited our search to locate those file extensions.

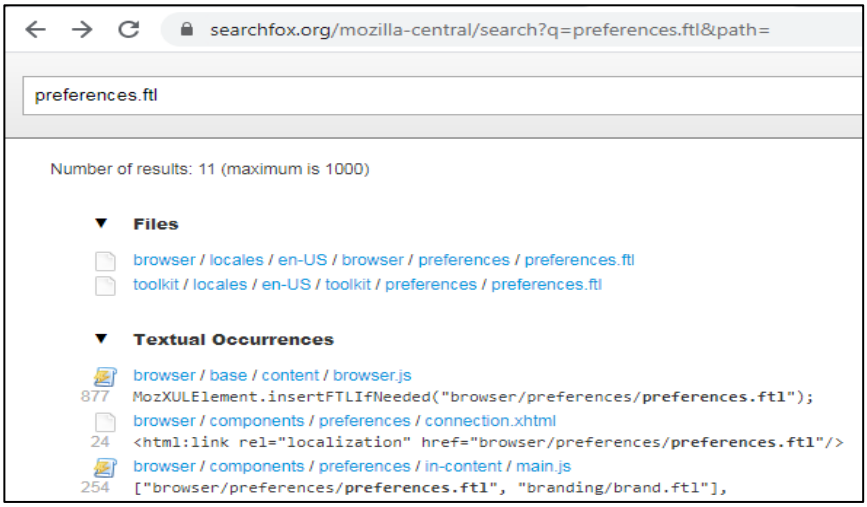


Fig. 19 Example results of searching a localization file (preferences.ftl) in Searchfox.

First, we consider files that contain the exact query string that can result in any file extension (e.g., .xul, .dtd, .ftl, .properties, .js, .xml, .cpp, etc.) as long as the content of the file matches our search criteria. Among the different strings that we use for our query matching are: comments in the code, string literals in localization files, or identifiers. From this initial result set, we store the .js, .cpp, .h, and .c file paths if there is any, and we do not rely on cross-referencing and navigating through the located source code files, instead we only use the name of

[44] https://searchfox.org/

the returned localization files[45] (i.e., .dtd, .ftl, and .properties file formats that are used to translate the user interface to different languages) or .xul files to search for other JavaScript or C/C++ paths, since these files contain references to GUI elements or text that is shown in the screencasts and their name is included in our target source code (Figure 19).Next, we further reduce the search space by only selecting source code files that contain the name of the located localization or .xul files and store the directories to these files. We stopped our guided search when a.) no more localization or .xul files could be found or b.) no additional JavaScript or C/C++ source code files were added to our search results. We then calculated the AP and RR values and compared them with the baseline results from the four RQs introduced earlier. Table 6 shows the AP and RR values of guided search for JavaScript and C/C++ code at the directory level. We also compared the highest median AP and RR values of the guided search against our feature location approach (Figure 20).

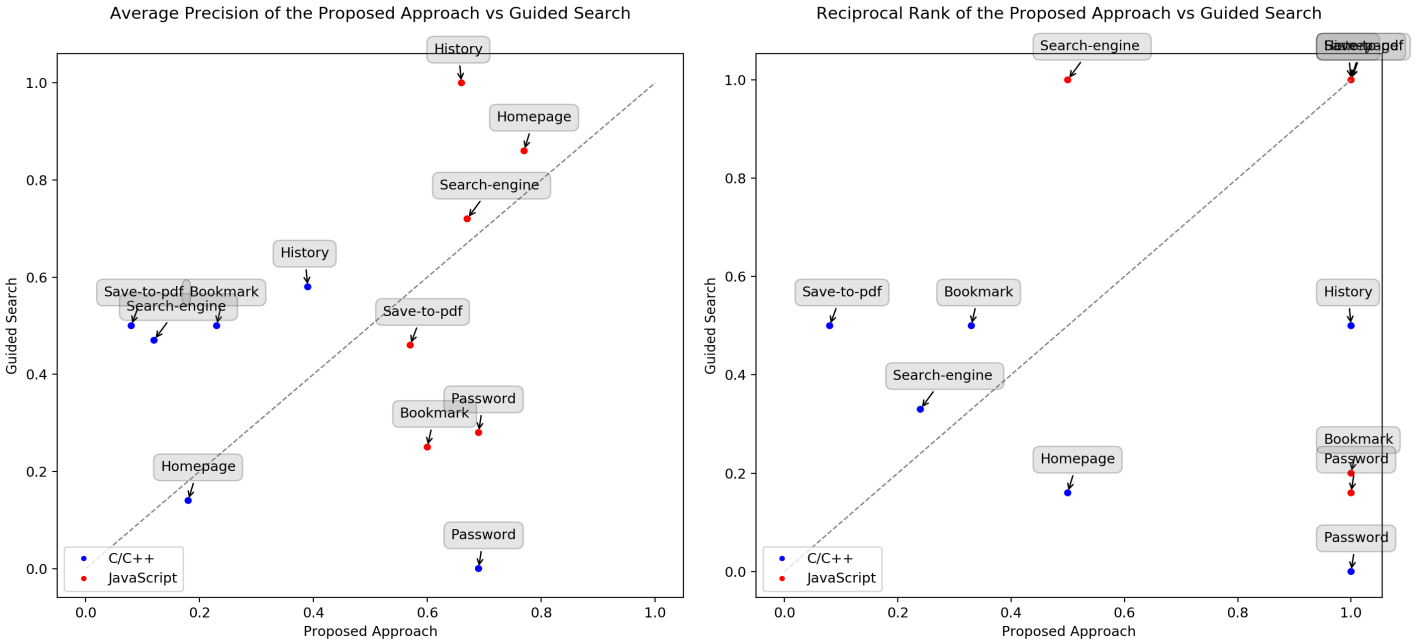


Fig. 20 Scatter plots of the best median AP and RR values of the proposed approach vs. the AP and RR values of guided search.

Table 6 Average Precision and Reciprocal Rank of guided search approach in Mozilla Firefox.

Scenario	Baseline	Guided Search	
		AP	RR
Import Bookmarks	C/C++	0.5	0.5
	JavaScript	0.25	0.2
Clear History	C/C++	0.58	0.5
	JavaScript	1	1
Set Homepage	C/C++	0.14	0.16
	JavaScript	0.86	1
Remove Passwords	C/C++	0	0
	JavaScript	0.28	0.16
Save-to-pdf	C/C++	0.5	0.5
	JavaScript	0.46	1
Default Search Engine	C/C++	0.47	0.33
	JavaScript	0.72	1

[45] https://mozilla-l10n.github.io/localizer-documentation/

Findings: The guided search and our feature location approach **both performed better for the frontend (which is developed in JavaScript) compared to the backend (which is developed in C/C++) results**. The scatter plot of the AP values shows that for the "*Default Search-engine*", "*Set Homepage*", and "*Clear History*" scenarios, the guided search outperformed our proposed approach. Two of these scenarios ("*Default Search-engine*" and "*Set Homepage*") contain a significant amount of noise due to having text that appears on web pages that is not relevant to these scenarios. While the guided approach performs better than our approach, an AP above 0.5 for our approach still indicates that the number of true positives is higher than the number of false positives in the top *X*/100 results returned for these three scenarios.

The AP values for C/C++ using our approach are mostly lower compared to the guided search results, except for the "*Remove Passwords*" and "*Set Homepage*" scenario where our approach outperforms the guided search. However, for 8 out of 12 RR values in the case of JavaScript and C/C++, our approach performed equal or better than the guided search. For the AP results, our approach outperformed guided search in 5 out of 12 cases. Furthermore, for all JavaScript directories, the best median AP value for our approach is above 0.5, which means that the number of true positives is larger than the number of false positives in the top 100 results.

Given that our approach attempts to locate only a seed or starting point for searching relevant source code artifacts, the median RR values (which correspond to the rank of the first true positive in the top hits), clearly indicate that our approach significantly outperforms guided search. Furthermore, using our automated approach, such relevant source code directories can be identified/extracted in advance and without the need to watch the screencast or manually interpret feature captions.

Conclusion: Although we only searched for JavaScript and C/C++ files/directories in our guided search and did not include any other call dependencies (files) that might be referenced within these source code files, we still found the guided search a time-consuming, manual process that needs to be repeated for each scenario. Even if our approach did not always outperform the guided search, especially for AP, our approach is automated and can therefore significantly speed up the location of relevant source code directories with an accuracy higher or similar to a guided search performed by a developer.

7. Discussion

Our case study results show that our current approach can link video content (application features) to guide developers and maintainers in locating source code related to features illustrated in the videos. Our approach works best for those features that are related to the frontend/GUI-related development of a project. We also found that speech is a very important component of a video and having high-quality speech is an essential information source.

In what follows, we present some general guidelines that can further improve the automated linking of features demonstrated in screencasts to their corresponding source code implementation. Our case study results confirm other researchers' findings (MacLeod, Bergen and Storey, 2017) about best practices in creating tutorial screencasts. The performance of our automated approach can be improved if developers and video creators follow these best practices:

1) Provide a transcript. Screencast creator should provide a transcript of their video and its content, eliminating the dependency on automated transcription services, which are still prone to errors.
2) In cases, when no video transcription is provided by the video creator, the screencasts should be presented using high-quality speech that is understandable, has less noise, and can be transcribed with high accuracy.
3) Recording high definition (HD) videos that have a readable text size (MacLeod, Bergen and Storey, 2017), which can improve the accuracy of the OCR's output.

4) Reducing noise in videos by showing only the features that are related to the topic that is being presented in the video and by minimizing the amount of time spent on showing non-relevant subject on the screen. This can be done by providing links to other screencasts that cover other topics (MacLeod, Bergen and Storey, 2017).

5) Providing complementary documents (MacLeod, Bergen and Storey, 2017) such as written documents or slides that can further improve the linking results.

6) Annotating or tagging screencasts using keywords that can help to clearly distinguish one video that covers a usage scenario from a video that covers another usage scenario (MacLeod, Bergen and Storey, 2017) .

7) How-to screencasts should include information about the version of the software application whose feature are being demonstrated to improve the location of the correct source code version.

Also, our findings confirm other software traceability researchers' findings (Gotel *et al.*, 2012) about programming guidelines and conventions that should be followed by developers to enable better source code localization:

8) Modifying the projects' naming conventions to reflect the high-level purpose or feature that a source code artifact (method, identifier, etc.) is implementing.

9) Include comments in the source code that describe what high-level features of the project are implemented by this source code.

10) Having separation of concerns and designing modular code by the use of design patterns (Leach, 2000).

8. Threats to Validity

This work is the first attempt to use crowd-based tutorial screencasts for feature location. Our case study results show that the approach can successfully locate relevant source code files in WordPress and source code directories in Firefox using a combination of the speech and GUI, or either data set. Nonetheless, there are some threats to external, internal and construct validity regarding the data sets and the methodology that need to be addressed in future work.

External Validity. In this work, 5 data sets of WordPress tutorial screencasts and 6 data sets of Mozilla Firefox tutorial screencasts, are used. This limited number of videos is due to our selection criteria, such as being uploaded in the same year (likely sharing the same WordPress or Firefox version), being of high quality, and containing both speech and GUI. Also, the curation of the data sets was time consuming, since the results from the text transcription and mined text from the image frames had to be manually verified. For the Firefox data sets, we faced additional challenges in finding tutorial videos with an English speaker narrator. We therefore also included screencasts without any speech. One approach to mitigate this threat would be applying the guidelines for creating screencasts (Section 7). Following these guidelines would significantly alleviate these challenges by providing a larger number of higher quality screencasts that are easier to process automatically. Also, automating the process of removing greetings and closing marks of a screencast can reduce the manual effort required for preparing the screencast data.

In this work, we mined only textual screencast information from speech and GUI elements. We also captured user actions through textual difference between every two subsequent image frames, and from the source code we only used textual information in the form of comments, identifiers, string literals and file names. Although these information sources already provide promising results, incorporating other information sources, such as the sequence of user events or for example call graphs from the source code should be considered for further improving the approach.

Our exploratory studies consider file and package/directory as granularity level for the WordPress and Mozilla Firefox evaluations respectively. In addition, our feature location approach could be refined to include also method- or even statement-level granularity in the analysis, to provide a more fine-grained analysis.

Construct Validity. Our approach can successfully retrieve the first true positive in the top 10 hits for WordPress, and top *X*/100 hits for Mozilla Firefox. However, the speech-to-text and OCR tools used in this work may have noise or false positives in their output. In our WordPress case study, we partially mitigate this problem by keeping only the words that have a confidence score or accuracy of 0.7 or more in speech documents. While using term weighting can alleviate the effect of such noise in our data sets, the analysis of the text from the transcription or code analysis could be further improved by more advanced speech-to-text or OCR approaches and by using language models and specific language parsers.

One other threat to construct validity is the use of LDA on short documents. Applying LDA approaches that are specifically designed for such documents (e.g., (Cheng *et al.*, 2014; Li *et al.*, 2016; Pedrosa *et al.*, 2017)) could improve the topic models.

Internal Validity. In the case of WordPress, we used two types of baselines to evaluate the performance of the approach. Since the creation of the *Unique* baseline requires us to compare files shared across scenarios, considering more scenarios can potentially reduce the size of unique data sets. In our evaluation, the exact size of the *Unique* baseline does not matter since we compare the performance across the *All* and *Unique* baselines instead of focusing only on the absolute performance values.

In case of Mozilla Firefox, the Gecko profiler does not provide the exact path to the source code files whose methods are executed. In addition, multiple runs of the same scenario are required to have the profiler capture a more complete execution trace for a specific scenario. Using other profiling approaches or tools (e.g., aspect-oriented programming, DTrace[46]) to capture the exact file paths should be considered as future work. Also, we used log likelihood values for different numbers of topics to determine the best number of topics to be used in our experiments. Future work should explore the sensitivity of our approach to the number of topics in more detail.

8. Conclusion

This paper presents a feature location approach that uses crowd-based screencasts for a given usage scenario to locate the scenario's source code implementation. We motivated our work, by highlighting results from a user survey which we conducted to evaluate the use of screencasts by developers We then present our methodology that takes as input textual information (e.g., comments, identifiers, and string literals) from the source code and audio and visual components of screencasts (i.e., speech and GUI text). We apply LDA to structure this information and rank the output seeds in the source code based on the similarity with the feature shown in the screencast. These ranked outputs provide a good starting point for further exploring the implementation of the screencasts' scenarios.

In a case study using 10 WordPress screencasts covering two different feature implementations, we evaluated the applicability of our approach in retrieving relevant results at the top 10 search results. We also performed another case study on 89 Mozilla Firefox screencasts, which cover 6 different Firefox features and in this case the ranking of relevant directories at the top *X*/100 hits. As part of our evaluation we also studied how modifying term frequencies, based on the importance of the terms, affects the feature location results. From our case studies, we observed that speech and image frame data are sufficient to perform our feature location approach. Our findings show that, in general, rebalancing of the data improves the quality of results.

[46] http://dtrace.org/blogs/about/

Future work should extend the data sets by including additional screencasts involving different scenarios and software applications. Also, while our work established that screencasts contain enough textual information, future work should extend our approach with other text retrieval-based approaches for feature location.

9. References

Adrian, K. *et al.* (no date) 'Software Cartography: thematic software visualization with consistent layout', *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3), pp. 191–210. doi: 10.1002/smr.414.

Ali, N. *et al.* (2012) 'Improving Bug Location Using Binary Class Relationships', in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 174–183. doi: 10.1109/SCAM.2012.26.

Asuncion, H. U., Asuncion, A. U. and Taylor, R. N. (2010) 'Software traceability with topic modeling', in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. Cape Town, South Africa: ACM Press, pp. 95–104. doi: 10.1145/1806799.1806817.

Bajracharya, S. K. and Lopes, C. V. (2012) 'Analyzing and mining a code search engine usage log', *Empirical Software Engineering*. Kluwer Academic Publishers, 17(4–5), pp. 424–466. doi: 10.1007/s10664-010-9144-6.

Baldi, P. F. *et al.* (2008) 'A theory of aspects as latent topics', in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. New York, New York, USA: ACM Press, pp. 543–562. doi: 10.1145/1449764.1449807.

Bao, L. *et al.* (2015) 'Reverse engineering time-series interaction data from screen-captured videos', *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., pp. 399–408. doi: 10.1109/SANER.2015.7081850.

Bao, L. *et al.* (2017) 'Extracting and analyzing time-series HCI data from screen-captured task videos', *Empirical Software Engineering*, 22(1), pp. 134–174. doi: 10.1007/s10664-015-9417-1.

Bao, L. *et al.* (2019) 'VT-Revolution: Interactive Programming Video Tutorial Authoring and Watching System', *IEEE Transactions on Software Engineering*, 45, pp. 823–838. doi: 10.1109/TSE.2018.2802916.

Barzilay, O., Treude, C. and Zagalsky, A. (2013) 'Facilitating Crowd Sourced Software Engineering via Stack Overflow', in *Finding Source Code on the Web for Remix and Reuse*. New York: Springer New York, pp. 1–19. doi: 10.1007/978-1-4614-6596-6.

Bassett, B. and Kraft, N. A. (2013) 'Structural information based term weighting in text retrieval for feature location', in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, pp. 133–141. doi: 10.1109/ICPC.2013.6613841.

Blei, D. M. *et al.* (2003) 'Hierarchical Topic Models and the Nested Chinese Restaurant Process', in *Proceedings of the 16th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press (NIPS'03), pp. 17–24.

Blei, D. M. (2012) 'Probabilistic topic models', in *Communications of the ACM*. ACM, pp. 77–84. doi: 10.1145/2133806.2133826.

Blei, D. M., Ng, A. Y. and Jordan, M. I. (2003) 'Latent dirichlet allocation', *The Journal of Machine Learning Research*. JMLR.org, 3, pp. 993–1022.

Brunelli, R. and Poggio, T. (1993) 'Face recognition: features versus templates', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10), pp. 1042–1052. doi: 10.1109/34.254061.

Campbell, J. C. *et al.* (2013) 'Deficient documentation detection: A methodology to locate deficient project documentation using topic analysis', *IEEE International Working Conference on Mining Software Repositories*. IEEE, Piscataway, NJ, USA, pp. 57–60. doi: 10.1109/MSR.2013.6624005.

Chen, T.-H., Thomas, S. W. and Hassan, A. E. (2016) 'A survey on the use of topic models when mining software repositories', *Empirical Software Engineering*, 21(5), pp. 1843–1919. doi: 10.1007/s10664-015-9402-8.

Cheng, X. *et al.* (2014) 'BTM: Topic modeling over short texts', *IEEE Transactions on Knowledge and Data Engineering*, 26(12), pp. 2928–2941. doi: 10.1109/TKDE.2014.2313872.

Cheriet, M. *et al.* (2007) *Character Recognition Systems: A Guide for Students and Practitioners*. Wiley-Interscience.

Cleland-Huang, J. *et al.* (2012) 'Breaking the big-bang practice of traceability: Pushing timely trace recommendations to project stakeholders', in *2012 20th IEEE International Requirements Engineering Conference (RE)*, pp. 231–240. doi: 10.1109/RE.2012.6345809.

Cleland-Huang, J. *et al.* (2014) 'Software traceability: trends and future directions', in *Proceedings of the on Future of Software Engineering - FOSE 2014*. New York, New York, USA: ACM Press, pp. 55–69. doi: 10.1145/2593882.2593891.

Deerwester, S. *et al.* (1990) 'Indexing by latent semantic analysis', *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6), pp. 391–407.

Dit, B. *et al.* (2013) 'Feature location in source code: A taxonomy and survey', *Journal of software: Evolution and Process*, 25(1), pp. 53–95. doi: 10.1002/smr.567.

Eddy, Brian P. and Kraft, Nicholas A. and Gray, J. (2018) 'Impact of structural weighting on a latent Dirichlet allocation–based feature location technique', *Journal of Software: Evolution and Process*, 30(1), p. e1892. doi: 10.1002/smr.1892.

Ellmann, M. *et al.* (2017) 'Find, Understand, and Extend Development Screencasts on YouTube', *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics - SWAN 2017*. New York, New York, USA: ACM Press, pp. 1–7. doi: 10.1145/3121257.3121260.

Escobar-Avila, J., Parra, E. and Haiduc, S. (2017) 'Text Retrieval-Based Tagging of Software Engineering Video Tutorials', in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 341–343. doi: 10.1109/ICSE-C.2017.121.

Gaffney Jr., J. E. (1981) 'Metrics in Software Quality Assurance', in *Proceedings of the ACM '81 Conference*. New York, NY, USA, NY, USA: ACM (ACM '81), pp. 126–130. doi: 10.1145/800175.809854.

Gotel, O. *et al.* (2012) 'The Grand Challenge of Traceability (v1.0)', in Cleland-Huang, J., Gotel, O., and Zisman, A. (eds) *Software and Systems Traceability*. London: Springer London, pp. 343–409. doi: 10.1007/978-1-4471-2239-5_16.

Gray, W. D. (2007) *Integrated Models of Cognitive Systems (Advances in Cognitive Models and Architectures)*. New York, NY, USA: Oxford University Press, Inc.

Grechanik, M. *et al.* (2010) 'A search engine for finding highly relevant applications', in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa: ACM Press, pp. 475–484. doi: 10.1145/1806799.1806868.

Hofmann, T. and Thomas (2001) 'Unsupervised Learning by Probabilistic Latent Semantic Analysis', *Machine Learning*. Kluwer Academic

Publishers, 42(1/2), pp. 177–196. doi: 10.1023/A:1007617005950.
Jiau, H. C. and Yang, F.-P. (2012) 'Facing up to the inequality of crowdsourced API documentation', *ACM SIGSOFT Software Engineering Notes*. ACM, 37(1), pp. 1–9. doi: 10.1145/2088883.2088892.
Jurafsky, D. and Martin, J. H. (2009) *Speech and Language Processing (2nd Edition)*. USA: Prentice-Hall, Inc.
Kagdi, H. and Maletic, J. I. (2007) 'Software Repositories : A Source for Traceability Links', *TEFSE/GCT 2007 - 4th International Workshop on Traceability in Emerging Forms of Software Engineering*, (APRIL 2002), pp. 32–39.
Kagdi, H., Maletic, J. I. and Sharif, B. (2007) 'Mining software repositories for traceability links', in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pp. 145–154. doi: 10.1109/ICPC.2007.28.
Keivanloo, I. (2013) 'Source Code Similarity and Clone Search'.
Keivanloo, I., Roy, C. K. and Rilling, J. (2014) 'SeByte: Scalable clone and similarity search for bytecode', *Science of Computer Programming*. Elsevier, 95, pp. 426–444. doi: 10.1016/J.SCICO.2013.10.006.
Khandwala, K. and Guo, P. J. (2018) 'Codemotion: Expanding the Design Space of Learner Interactions with Computer Programming Tutorial Videos', in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale - L@S '18*. London, United Kingdom: ACM Press, pp. 1–10. doi: 10.1145/3231644.3231652.
Kuhn, A., Ducasse, S. and Gîrba, T. (2007) 'Semantic clustering: Identifying topics in source code', *Information and Software Technology*. Elsevier, 49(3), pp. 230–243. doi: 10.1016/J.INFSOF.2006.10.017.
Kuhn, A., Loretan, P. and Nierstrasz, O. (2012) 'Consistent Layout for Thematic Software Maps'. doi: 10.1109/WCRE.2008.45.
Leach, R. J. (2000) *Introduction to Software Engineering*. Boca Raton, FL, USA: CRC Press, Inc.
Li, C. *et al.* (2016) 'Topic Modeling for Short Texts with Auxiliary Word Embeddings', in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval - SIGIR '16*. Pisa, Italy: ACM Press, pp. 165–174. doi: 10.1145/2911451.2911499.
Lukins, S. K., Kraft, N. A. and Etzkorn, L. H. (2010) 'Bug localization using latent Dirichlet allocation', *Information and Software Technology*. Elsevier B.V., 52(9), pp. 972–990. doi: 10.1016/j.infsof.2010.04.002.
MacLeod, L., Bergen, A. and Storey, M.-A. (2017) 'Documenting and sharing software knowledge using screencasts', *Empirical Software Engineering*, 22(3), pp. 1478–1507. doi: 10.1007/s10664-017-9501-9.
MacLeod, L., Storey, M.-A. and Bergen, A. (2015) 'Code, Camera, Action: How Software Developers Document and Share Program Knowledge Using YouTube', *2015 IEEE 23rd International Conference on Program Comprehension (ICPC)*. IEEE, Piscataway, NJ, USA. doi: 10.1109/ICPC.2015.19.
Manning, C. D., Raghavan, P. and Schütze, H. (2008) *Introduction to Information Retrieval*. USA: Cambridge University Press.
Marcus, A. *et al.* (2004) 'An information retrieval approach to concept location in source code', in *11th Working Conference on Reverse Engineering*. USA: IEEE Comput. Soc, pp. 214–223. doi: 10.1109/WCRE.2004.10.
Mcauliffe, J. D. and Blei, D. M. (2008) 'Supervised Topic Models', in Platt, J. C. et al. (eds) *Advances in Neural Information Processing Systems 20*. Curran Associates, Inc., pp. 121–128. Available at: http://papers.nips.cc/paper/3328-supervised-topic-models.pdf.
Mohorovičič, S. (2012) 'Creation and use of screencasts in higher education', *MIPRO 2012 - 35th International Convention on Information and Communication Technology, Electronics and Microelectronics - Proceedings*, pp. 1293–1298.
Moslehi, P., Adams, B. and Rilling, J. (2016) 'On mining crowd-based speech documentation', in *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. Austin, Texas: ACM Press, pp. 259–268. doi: 10.1145/2901739.2901771.
Moslehi, P., Adams, B. and Rilling, J. (2018) 'Feature Location using Crowd-based Screencasts', in *Proceedings of the 15th IEEE Working Conference on Mining Software Repositories (MSR)*. Gothenburg, Sweden, pp. 192–202. doi: 10.1145/3196398.3196439.
Moslehi, P., Rilling, J. and Adams, B. (2020) 'Adoption of Crowd-based Software Engineering Tutorial Screencasts'. Available at: https://mcislab.github.io/publications/2020/OnTheUseOfMultimediaDocumentation.pdf.
Nasehi, S. M. *et al.* (2012) 'What makes a good code example?: A study of programming Q&A in StackOverflow', in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Piscataway, NJ, USA, pp. 25–34. doi: 10.1109/ICSM.2012.6405249.
Nguyen, A. T. *et al.* (2012) 'Duplicate bug report detection with a combination of information retrieval and topic modeling', in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. Essen, GermanyUSA: ACM Press, pp. 70–79. doi: 10.1145/2351676.2351687.
Nixon, M. S. and Aguado, A. S. (2012a) 'Chapter 5 - High-level feature extraction: fixed shape matching', in Nixon, M. S. and Aguado, A. S. (eds) *Feature Extraction and Image Processing for Computer Vision (Third edition)*. Third edit. Oxford: Academic Press, pp. 217–291.
Nixon, M. S. and Aguado, A. S. (2012b) 'Chapter 7 - Object description', in Nixon, M. S. and Aguado, A. S. (eds) *Feature Extraction and Image Processing for Computer Vision (Third edition)*. Third edit. Oxford: Academic Press, pp. 343–397.
Ott, J. *et al.* (2018) 'A Deep Learning Approach to Identifying Source Code in Images and Video', *International Conference on Mining Software Repositories (MSR)*, pp. 376–386. doi: 10.1145/3196398.3196402.
Parnin, C. *et al.* (2012) 'Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow', *Georgia Tech Technical Report*. Available at: http://dl.acm.org/citation.cfm?id=1806855%5Cnhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.371.6263&rep=rep1&type=pdf.
Parra, E., Escobar-Avila, J. and Haiduc, S. (2018) 'Automatic tag recommendation for software development video tutorials', in *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*. Gothenburg, Sweden: ACM Press, pp. 222–232. doi: 10.1145/3196321.3196351.
Pedrosa, G. *et al.* (2017) 'Topic Modeling for Short Texts with Co-occurrence Frequency-Based Expansion', *Proceedings - 2016 5th Brazilian Conference on Intelligent Systems, BRACIS 2016*, pp. 277–282. doi: 10.1109/BRACIS.2016.058.
Pham, R. *et al.* (2013) 'Creating a shared understanding of testing culture on a social coding site', in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, pp. 112–121. doi: 10.1109/ICSE.2013.6606557.
Poche, E. *et al.* (2017) 'Analyzing User Comments on YouTube Coding Tutorial Videos', in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. Buenos Aires, Argentina, pp. 196–206. doi: 10.1109/ICPC.2017.26.
Ponzanelli, L. *et al.* (2016) 'Too long; didn't watch!: extracting relevant fragments from software development video tutorials', in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, pp. 261–272. doi: 10.1145/2884781.2884824.
Ponzanelli, L. *et al.* (2019) 'Automatic Identification and Classification of Software Development Video Tutorial Fragments', *IEEE Transactions on Software Engineering*, 45, pp. 464–488. doi: 10.1109/TSE.2017.2779479.

Ramage, D. *et al.* (2009) ‘Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora’, in *EMNLP 2009 - Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: A Meeting of SIGDAT, a Special Interest Group of ACL, Held in Conjunction with ACL-IJCNLP 2009*. Singapore, pp. 248–256.
van der Spek, P., Klusener, S. and van de Laar, P. (2008) ‘Towards Recovering Architectural Concepts Using Latent Semantic Indexing’, in *2008 12th European Conference on Software Maintenance and Reengineering*, pp. 253–257. doi: 10.1109/CSMR.2008.4493321.
Storey, M.-A. *et al.* (2014) ‘The (R) Evolution of social media in software engineering’, *Proceedings of the on Future of Software Engineering - FOSE 2014*, pp. 100–116. doi: 10.1145/2593882.2593887.
Subramanian, S., Inozemtseva, L. and Holmes, R. (2014) ‘Live API documentation’, in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. Hyderabad, India, pp. 643–652. doi: 10.1145/2568225.2568313.
Thomas, S. W. *et al.* (2010) ‘Validating the Use of Topic Models for Software Evolution’, in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, pp. 55–64. doi: 10.1109/SCAM.2010.13.
Thomas, S. W. (2012) *Mining Unstructured Software Repositories Using IR Models*. Queen’s University.
Turk, D., France, R. and Rumpe, B. (2014) ‘Limitations of Agile Software Processes’, abs/1409.6, pp. 43–46.
Wallach, H. M. (2006) ‘Topic Modeling: Beyond Bag-of-words’, in *Proceedings of the 23rd International Conference on Machine Learning*. Pittsburgh, Pennsylvania, USA: ACM (ICML ’06), pp. 977–984. doi: 10.1145/1143844.1143967.
Wang, X., McCallum, A. and Wei, X. (2007) ‘Topical N-grams: Phrase and topic discovery, with an application to information retrieval’, *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 697–702. doi: 10.1109/ICDM.2007.86.
Wells, J., Barry, R. M. and Spence, A. (2012) ‘Using Video Tutorials as a Carrot-and-Stick Approach to Learning’, *IEEE Transactions on Education*, 55(4), pp. 453–458. doi: 10.1109/TE.2012.2187451.
Yadid, S. and Yahav, E. (2016) ‘Extracting code from programming tutorial videos’, in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2016*. Amsterdam, Netherlands: ACM Press, pp. 98–111. doi: 10.1145/2986012.2986021.